REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
03-06-2003	Final Report	15-09-2000 - 31-05-2003

4. TITLE AND SUBTITLE
Mechanizing Arithmetic II

5a. CONTRACT NUMBER
MDA904-00-C-3662

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Golden, Jeff
Hunt, Jr., Warren A.
Krug, Robert
Moore, J. Strother

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The University of Texas at Austin
PO Box 7726
Austin, TX 78713-7726

8. PERFORMING ORGANIZATION REPORT NUMBER
200001461

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Maryland ProcurementOffice
6800 Savage Road, Suite 6720
Ft. Meade, MD 20755-6720

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The primary goal of this project has been to imporve the degree of automation in the ACL2 when proving theorems about computer programs. We have paid particular attention to fixed-width integer arithmetic and the Java Virtual Machine (JVM). Our work has proceeded on three fronts.
*We have begun developing an approach for automating much of the detail work involved in applying the "ACL2 method" for proving theorems about low-level programs. Work completed includes a prototype tool targeting the language of the JVM, and a preliminary JVM-specific lemma library designed to facilitate reasoning about refined JVM models.
*We have improved the ACL2 theorem prover itself. We have also developed an improved library of arithmetic lemmas.
*We have begun work in cooperation with the Destiny Project. Destiny is an automated program verifier under development by the government. We have developed a first version of a tool that reads Destiny's XML output and converts it to ACL2.

15. SUBJECT TERMS
ACL2, proving theorems, JVM models, arithmetic lemmas, Destiny

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Warren A. Hunt, Jr.
U	U	U	UU	111	19b. TELEPHONE NUMBER *(Include area code)* 512-471-7316

Report of the Project for Mechanizing Arithmetic

Jeff Golden
Warren A. Hunt, Jr.
Robert Krug
J Strother Moore

June 3, 2003

Abstract

The primary goal of this project has been to improve the degree of automation in ACL2 when proving theorems about computer programs. We have paid particular attention to fixed-width integer arithmetic and the Java Virtual Machine (JVM). Our work has proceeded on three fronts.

- We have begun developing an approach for automating much of the detail work involved in applying the "ACL2 method" for proving theorems about low-level programs. Work completed includes a prototype tool targeting the language of the JVM, and a preliminary JVM-specific lemma library designed to facilitate reasoning about refined JVM models.
- We have improved the ACL2 theorem prover itself. We have also developed an improved library of arithmetic lemmas.
- We have begun work in cooperation with the Destiny Project. Destiny is an automated program verifier under development by the government. We have developed a first version of a tool which reads Destiny's XML output and converts it into ACL2. We have begun work on libraries which will allow us to verify the converted output within ACL2.

Contents

1 Introduction **5**
- 1.1 Overview of the Project 5
- 1.2 Plan of the Paper 6

2 Mechanical Generation of Verifiably Correct Program Abstractions **9**
- 2.1 Introduction 9
- 2.2 Motivation 10
- 2.3 The addThree Method 12
- 2.4 Characterization Functions and Correspondence Proofs 14
- 2.5 Our Approach 17

3 Enhancements to ACL2 **19**
- 3.1 Introduction 19
- 3.2 Some Minor Improvements 20
  - 3.2.1 Removing Linear-alias Rules 20
  - 3.2.2 Parent Trees 20
  - 3.2.3 Normalizing Polynomials 21
  - 3.2.4 A Better Poly-memberp 21
- 3.3 Meeting Users Expectations 22
  - 3.3.1 Regularizing Equality substitutions 22
  - 3.3.2 Eliminating Unwanted Hypothesis Elimination 22
  - 3.3.3 Forward Chaining and Linear Arithmetic 24
- 3.4 Command and Control 27
  - 3.4.1 Backchain Limits 27
  - 3.4.2 Meta Rules 29
  - 3.4.3 Bind-free 35
- 3.5 Improved Context Maintenance 38
- 3.6 Linear Arithmetic and Type Reasoning 39
  - 3.6.1 Introduction 39
  - 3.6.2 Disentangling and Reintegrating 39
  - 3.6.3 Benefits Accrued 40
- 3.7 Nonlinear Arithmetic 40
  - 3.7.1 Motivational example 40

3.7.2 Theory 41
3.7.3 Linear Example 41
3.7.4 Non-linear Example 42

4 Library Development **44**
4.1 Introduction 44
4.2 Library Design 44
4.3 Conclusion 45

5 The Destiny Project **46**
5.1 Destiny Background 46
5.2 Reading Destiny's XML 47
5.3 Verifying Destiny's Rulebase 47

A Three Sample Lemmatta **49**
A.1 Example One 49
A.2 Example Two 50
A.3 Example Three 52

B Legato's Multiplier **54**
B.1 Introduction 54
B.2 Proof Script 54

C Extended Metafunctions Examples **62**

D Meta-Integerp **64**
D.1 Introduction 64
D.2 The Code Listing 65
D.3 Conclusion 71

E Bind-Free Examples **72**
E.1 Example One — Cancel a Common Factor 72
E.2 Example Two — Pull Integer Summand out of Floor 73
E.3 Example Three — Simplify Terms Such as `(equal (+ a (* a b)) 0)` 74

F Destiny XML to ACL2 Example **77**
F.1 XML Rulebase Snippet 77
F.2 Tool Output 82
F.3 The Final ACL2 Rules 83

G Assume-true-false **84**
G.1 Mini-essay on Assume-true-false-if 84

H README **88**
H.1 How to Prove Theorems Using ACL2 88
H.1.1 Introduction 88
H.1.2 How to Use the Theorem Prover — "The Method" 88
H.1.3 Structured Theory Development. 90
H.1.4 A Few Notes. 93
H.2 How to Use the Arithmetic-4 Books 95
H.2.1 Choosing a Normal Form. 95
H.2.2 Mini-theories. 98

I Non-Linear Arithmetic **99**
I.1 abstract 99
I.2 Introduction 99
I.2.1 Related Work and Plan of the Paper 101
I.3 Background 102
I.3.1 Type Reasoning and Polys from Type-set 103
I.3.2 Linearization 103
I.4 Linear Arithmetic 104
I.4.1 The Linear Arithmetic Algorithm 104
I.4.2 An Example 105
I.5 Linear Lemmas 105
I.5.1 The Linear Lemmas Algorithm 106
I.5.2 An Example 106
I.6 Linear Lemmas Revised 107
I.6.1 Exploded Pot Labels 107
I.6.2 Inverse Polys and Bounds Polys 108
I.6.3 The Revised Linear Lemmas Algorithm 108
I.6.4 An Example — Part I 109
I.7 Nonlinear Arithmetic 109
I.7.1 Deal-with-product and Deal-with-factor 109
I.7.2 Deal-with-division 110
I.7.3 The nonlinear Arithmetic Algorithm 111
I.8 Conclusion 111

Chapter 1

Introduction

1.1 Overview of the Project

This project has been devoted to raising the level of automation in ACL2 when proving theorems about computer programs. Our work can be broadly divided into the following categories.

Automation of the "ACL2 Method": Our formal model of the JVM, M5, has served as a basis for developing a preliminary JVM-specific lemma library to facilitate reasoning about refined JVM models, and also as a means of generating realistic proof obligations for use in benchmarking improvements to the theorem prover, itself. More than this, the model has allowed us to explore additional opportunities for automation via the mechanical generation of verifiably correct program abstractions. This work helps automate an established ACL2 method for verifying the correctness of low-level programs for which high assurance is required. Preliminary results are described in Chapter 2. An appendix section contains commented sample output of a prototype tool targeting the language of the JVM.

Improvements to ACL2 Itself: We have made a substantial number of improvements to ACL2 itself. These enhancements decrease user effort when using ACL2. All of them were motivated by specific problems encountered while using ACL2. They are discussed in Chapter 3, but we provide a brief overview of some of the highlights below. See Appendices A and B for some example lemmas which we can now prove more easily.

- The introduction of a nonlinear arithmetic reasoning package. We discuss this briefly in Section 3.7 and more fully in Appendix I. This new facility allows the unassisted proof of many lemmas which were difficult to prove before.
- The introduction of two new types of rules — extended meta rules and bind-free rules. These rules are discussed in Section 3.4. They

allow one to direct ACL2's operations in ways which were not previously possible. When we were first trying to write an improved arithmetic library, we found that some of the rules which we wished to write could not be expressed in ACL2 at that time. We developed these rule types in response to this situation.

- The elimination of certain types of unexpected and undesirable behavior on the part of ACL2. See Section 3.3. These changes were all made in response to user complaints about unexpected behavior. While the changes are not all substantive, they eliminate surprises and make ACL2 less irritating to use.
- Improved context maintenance. See Section 3.5. In certain situations during rewriting, ACL2 was ignoring some of the available contextual information. We have fixed this.

Development of an Improved Library: We have developed an improved library of lemmas about arithmetic functions. Some of this work involved carefully examining existing libraries to determine their strengths or weaknesses. However, a majority of our time was spent experimenting with "what if" scenarios. As hinted above, we were often frustrated with the expressiveness of ACL2 rule-classes. We went through many iterations of adding experimental facilities to ACL2, writing rules using these new facilities, and debugging our initial ideas. The result of all this is a library which allows much simpler proofs of the users arithmetic lemmas than existed before. This work is discussed in Chapter 4. As above, see Appendices A and B for some example lemmas which we can now prove more easily.

Support for Destiny: Destiny is an automated program verifier for Java currently under development by the government. Many of Destiny's operations are controlled by rules stored in an XML rule base. In addition, Destiny may produce XML output which requires verification by external theorem provers such as ACL2. We have developed a first version of a tool which reads Destiny's XML rule base or output, and converts it into ACL2. We have also begun work on libraries which will allow us to verify these converted rules in ACL2. See Chapter 5.

1.2 Plan of the Paper

This report is organized as follows.

- In Chapter 2, *Mechanical Generation of Verifiably Correct Program Abstractions*, we discuss development of a method aimed at providing the ACL2 user with a simpler starting point for analysis when verifying low-level programs. This section also describes a prototype tool targeting the language of the Java Virtual Machine.

- In Chapter 3, *Enhancements to ACL2*, we describe several enhancements we have made to ACL2.

  This is broken down into the following subsections

  - *Some Minor Improvements* We have made several minor improvements to ACL2 which are, generally, invisible to the user. We present a representative sample of these changes here.
  - *Meeting Users Expectations* The changes described in this section were all motivated by complaints from users. While they are not all substantive, they eliminate surprises and make ACL2 less irritating to use.
  - *Command and Control* The changes described in this section were motivated by desires to desires to write rules which had no natural expression in ACL2 at the time. They add facilities to control the operations of ACL2 more precisely.
  - *Improved Context Maintenance* ACL2 stores the "context" under which rewriting occurs in a structure called the type-alist. Previously, under certain circumstances the type-alist was not augmented with new contextual information correctly. We discuss our solution to this.
  - *Linear Arithmetic and Type Reasoning* We discuss some work on low-level aspects of the integration of linear arithmetic and type reasoning.
  - *Non-Linear Arithmetic* In this section, we present a brief overview of non-linear arithmetic. In Appendix I, we give a more detailed presentation.

- In Chapter 4, *Library Development*, we discuss our work developing an improved library of arithmetic lemmas.

- In Chapter 5, *The Destiny Project*, we discuss our work in support of the Destiny project.

- In Appendix A, *Three Sample Lemmatta*, we present three sample lemmatta we can now prove more easily.

- In Appendix B, *Legato's Multiplier*, we present an ACL2 proof script which solves a challenge posed by Bill Legato — to prove that a program written for the Mostek 6502 microprocessor correctly implements multiplication. The program is described in "A Weakest Precondition Model for Assembly Language Programs" by Bill Legato, dated June 19, 2000.

- In Appendix C, *Extended Metafunctions Examples*, we present some simple examples of the use of metafunctions.

- In Appendix D, *Meta-Integerp*, we include the source for a larger example extended meta rule, `meta-integerp-correct`. This rule is used in our arithmetic library.
- In Appendix E, *Bind-Free Examples*, we give examples of the use of bind-free hypotheses from the perspective of a user interested in reasoning about arithmetic.
- In Appendix G, *Assume-true-false*, we give a mini-essay on our work to improve the internal ACL2 function `assume-true-false`. This essay was originally written to document the new code, and so is directed towards someone who is reading it in company with the code. It appears in the file `type-set-b.lisp` and can be found by searching for the string "mini-essay".
- In Appendix H, *README*, we give the README from our arithmetic library, Arithmetic-4.
- In Appendix I, *Non-Linear Arithmetic*, we reproduce our paper on non-linear arithmetic.

Chapter 2

Mechanical Generation of Verifiably Correct Program Abstractions

2.1 Introduction

The work described in this chapter aims to provide the ACL2 user with an easier starting point for analysis when verifying the correctness of sequential, assembly or machine language programs. More specifically, we describe a preliminary approach for mechanically generating an ACL2 function characterizing "what such a program computes." The sense in which the program computes the function is made precise in what is called the correspondence theorem for the pair, and the goal is to generate not only a definition of the characterization function, but also auxiliary definitions and lemmas sufficient for ACL2 to certify this theorem.

As a first-class citizen of the ACL2 logic, a characterization function is amenable to formal analysis using the theorem prover. Mechanical generation of its definition and events sufficient to establish its legitimacy as a characterization permits the ACL2 user to approach the task of verifying properties of a low-level program by "doing what he or she does best," namely, driving the theorem prover to establish properties of an ACL2 function: a correspondence theorem for a function/program pair configures the theorem prover to automatically deduce from any established property of the function, a corresponding property of the program.

As a secondary benefit, the ACL2 user is shielded from much of the "detail work" involved in verifying a low-level program because the kind of automatic application of language and domain-specific knowledge that happens during the course of a successful proof is harnessed in simplifying the characterization function's definition.

The automation described here is not possible in the general case, but experience leads us to believe that it can be accomplished for a variety of arithmetic-intensive low-level programs.

We present motivation and background for this work in Sections 2.2 and 2.3, and details of the formalization in Section 2.4. In Section 2.5, we explain our preliminary mechanization in the context of generating verifiably correct characterization functions for Java Virtual Machine methods.

2.2 Motivation

The present work helps automate an established methodology for verifying the correctness of low-level programs depicted in Figure 2.1. Input enters the tool flow at the top left, in the form of a high-level source program. An off-the-shelf compiler translates this program into the language of a machine of interest, say a particular microprocessor or virtual machine, and a special-purpose program translates the resulting low-level code into a representation suitable for theorem proving. The result of this completely automatic portion of the tool flow is a formal object "understood" by the machine model depicted at the lower-right corner of the figure. This model, written in the logic of the theorem prover, formally defines the semantics of the low-level language by interpreting each instruction-like object in the program's formal representation as inducing an effect on modeled machine state. This style of language modeling is well-suited to formalizing low-level languages, whose "official" semantics are often specified operationally, and plays an important role in ensuring model accuracy.

Generation of the program-representing formal object marks what has typically been the end of the fully automatic portion of the tool flow. At this point an expert enters the picture, employing knowledge of the low-level language and expertise in using the theorem prover to formulate the goal correctness theorem as well as intermediate lemmas sufficient to make a proof of the goal theorem "obvious" to ACL2. It should be noted that because the final correctness theorem "speaks" directly about the modeled low-level program, the guarantee provided by ACL2 certification is completely independent of the correctness of the source program and compiler. This feature of the methodology allows for flexibility as well as assurance in that the upper left portion of the figure can be instantiated with any high level language for which there is a compiler with a back-end targeting the modeled machine.

The feasibility of the methodology described, thus far, is limited by its reliance upon an ACL2 user willing and able to manage details of the low-level language and details of the language interpreter, but this is more a symptom of the "state of the art" than an inherent limitation. The last fully automatic step, depicted as the rightward-leading arrow across the bottom of Figure 2.1, has the express purpose of mapping programs into the formal system in a way that makes the correspondence between informal language specification and formal language model open to inspection. To accomplish this goal, the tool implementing this translation must forgo many opportunities for "improvement,"

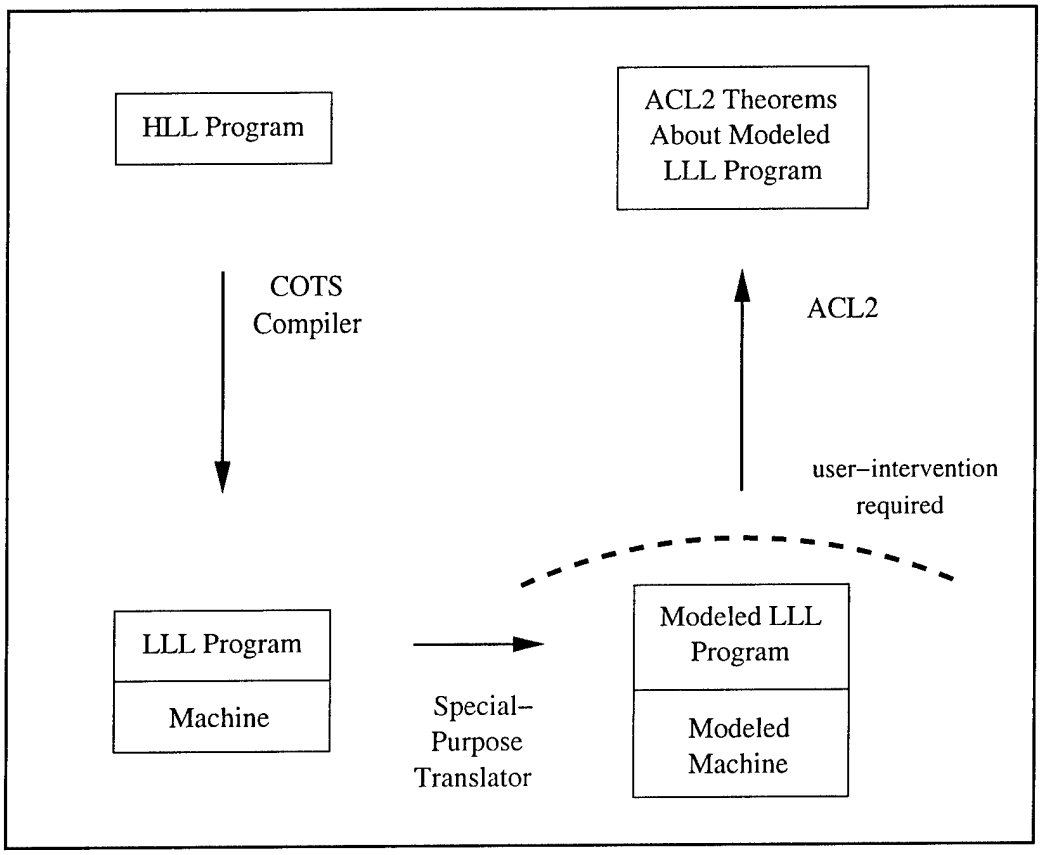


Figure 2.1: A Methodology

including not just the removal of the low-level language syntax, but also simplification of operational symbolic expressions, and indeed elimination of the entire language interpreter device. The present automation work pursues these opportunities aggressively, but leverages the previous work by using the formal object produced in this last step as a kind of formal checkpoint with respect to which simplifications are verified to be correct.

2.3 The addThree Method

To make our discussion more concrete, consider the problem of establishing conditions under which a non-negative integer result is returned by a static JVM method, addThree, whose instructions are given below. Following the tool flow associated with the JVM instantiation of the methodology outlined in Figure 2.2, we first list the Java source code from which the JVM method derives:

```
public static int addThree(int n) {
    n++;
    n++;
    n++;
    return n;
}
```

Sun's compiler, javac, produces a code array for addThree consisting of the following byte-code sequence, as displayed by Sun's class file disassembler, javap:

```
iinc 0 1
iinc 0 1
iinc 0 1
iload_0
ireturn
```

which the jvm2acl2 tool translates into the following lisp constant:

```
'((IINC 0 1)
  (IINC 0 1)
  (IINC 0 1)
  (ILOAD_0)
  (IRETURN))
```

Roughly speaking, addThree receives its input, a 32-bit two's complement integer (type int), in slot 0 of the local variable array associated with an invocation of the method. The value in slot 0 is then incremented by one, three times in succession, loaded onto the invocation's operand stack, and returned.

The name addThree is a misnomer in that certain inputs produce outputs that are not three greater in the standard sense. For example, while an input of 3

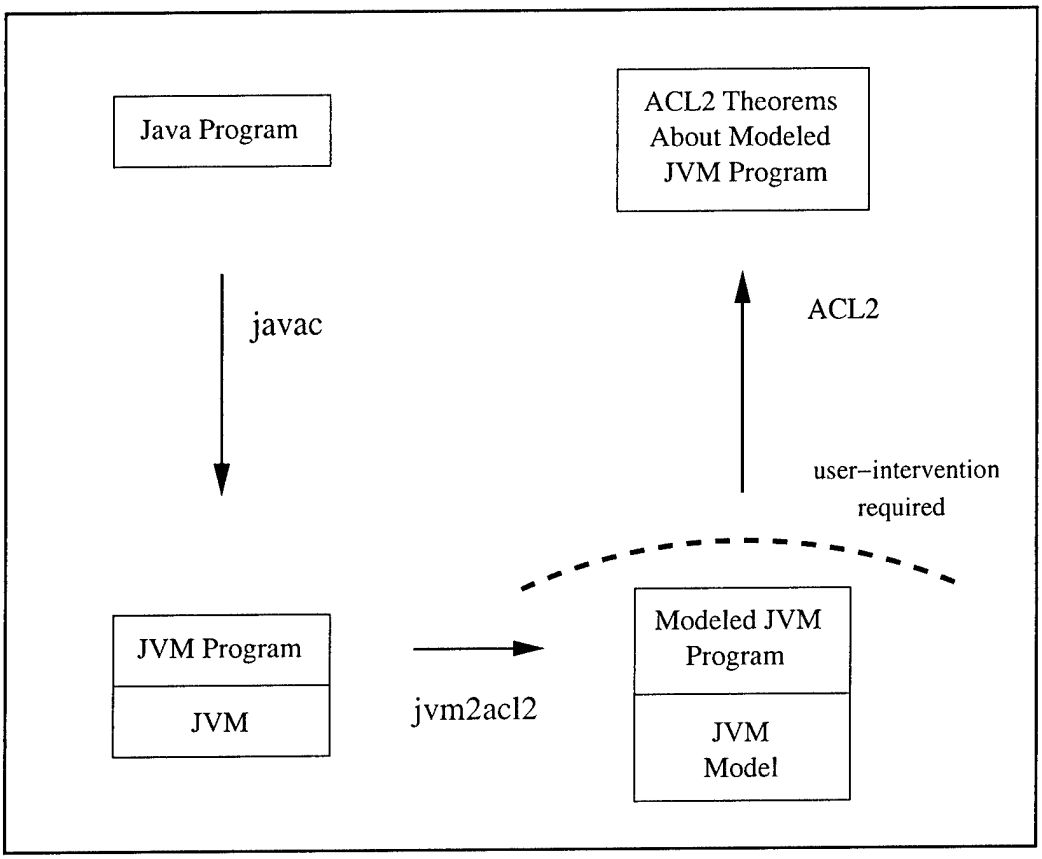


Figure 2.2: JVM Instantiation of a Methodology

produces an output of 6, an input of 2147483647, the largest int value, produces -2147483646. The anomalous behavior derives from Sun's specification that fixed-width integer operations like iinc produce a result equal to

> "the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type int."

These details clearly bear upon our example problem and are made precise by our model of the JVM, M5, which interprets each byte-code-like object in the list produced by jvm2acl2 as inducing an effect on modeled machine state patterned after the corresponding English-language specification in Sun's specification. While this method of language modeling is well-suited to formalizing Sun's operational specification, and allows for model validation via inspection, without further automation, the operational details that play such an important role in shortening the necessary "leap of faith" between informal language specification and formal language model become a liability when verifying properties of modeled programs. Managing them involves the user unnecessarily in details of the machine model as well as details of the modeling device.

Rather than solve our addThree problem here, illustrating typical user involvement in managing such details, we defer a solution until the next section, where we revisit the problem from the simpler, more familiar, starting point for analysis that the present automation work aims to provide. For some of the flavor of the manual detail work made unnecessary by our approach, the reader is referred to Section 2.5, where we discuss our mechanization.

2.4 Characterization Functions and Correspondence Proofs

Consider now the problem of establishing conditions under which a non-negative integer result is returned by the following ACL2 function:

```
(defun addThree-abstract-semantics (n)
  (if (not (intp n))
      n
    (int-fix (+ 3 n))))
```

where int-fix models the fixing operation described in the previous section, and is defined as follows:

```
(defun int-fix
  (let ((temp (mod (ifix x) (expt 2 32))))
    (if (< temp (expt 2 (1- 32)))
        temp
      (- temp (expt 2 32)))))
```

As its name suggests, the function addThree-abstract-semantics is intended to be used as the "meaning" of the addThree byte-code method described in the previous section. However, we don't put forward its definition as merely an educated guess. The goal is to generate not only a characterization function's definition but also a theorem like the following which asserts that the mathematical function represented by addThree-abstract-semantics is the function computed by the byte-code method, under assumptions made explicit in the statement of the theorem:

```
(defthm addThree-correspondence
  (implies (poised-to-invoke-addThree th s n)
           (equal
            (run (addThree-sched th n) s)
            (modify th s
                    :pc (+ 3 (pc (top-frame th s)))
                    :stack
                    (push (addThree-abstract-semantics n)
                          (pop (stack (top-frame th s))))))))
```

where poised-to-invoke-addThree is defined as follows

```
(defun poised-to-invoke-addThree (th s n)
  (and (equal (status th s) 'SCHEDULED)
       (equal (next-inst th s)
              '(invokestatic "AddThreeDemo" "addThree" 1))
       (equal n (top (stack (top-frame th s))))
       (intp n)
       (equal (lookup-method "addThree" "AddThreeDemo" (class-table s))
              '((IINC 0 1)
                (IINC 0 1)
                (IINC 0 1)
                (ILOAD_0)
                (IRETURN)))))
```

One of the guarantees provided by this theorem is that if n denotes a JVM int value, and we run the addThree method on n to completion, the result returned on the operand stack is equal to the value of addThree-abstract-semantics at n. The theorem is involved partly because, for pragmatic reasons, we set it up to assert more than just this correspondence of results. So that we can more easily apply this theorem in the context of verifying a method invoking addThree, we ensure that the theorem specifies not only the result computed but also that result's placement in the resulting JVM state, as well as a complete specification of all "side effects" of the computation.

This correspondence theorem makes precise the relationship between the two versions of our addThree problem, and to the extent that it can be generated (and certified) automatically, the ACL2 user is free to pursue a solution to the original problem about the modeled low-level program by establishing a property

of the characterization function. Our original problem has been reduced to defining a one-place predicate P such that the following becomes a certified theorem.

```
(defthm addThree-sufficient-precondition-for-non-negative-result-template
  (implies (P n)
           (<= 0 (addThree-abstract-semantics n))))
```

ACL2 is configured to generate from any proof of an instantiation of the theorem above, a proof of a corresponding theorem about the byte code method. In particular, from the theorem

```
(defthm addThree-sufficient-precondition-for-non-negative-result
  (implies
    (and (intp n)
         (<= 0 n)
         (<= n (- *largest-int-value* 3)))
    (<= 0 (addThree-abstract-semantics n))))
```

where intp is a predicate defined to recognize JVM int values, ACL2 deduces

```
(defthm addThree-main-result
  (implies (and (poised-to-invoke-addThree th s n)
                (<= 0 n)
                (<= n (- *largest-int-value* 3)))
           (<= 0
             (top
               (stack
                 (top-frame th
                            (run (addThree-sched th n) s)))))))
```

Notice that the predicate

```
(and (intp n)
     (<= 0 n)
     (<= n (- *largest-int-value* 3)))
```

which "solves" our example problem, is in fact stronger than required. After examining details of our mechanization in the next section, we revisit our addThree problem one more time to illustrate how our approach can be naturally extended to generate verifiably correct weakest preconditions with respect to user-supplied properties of interest. For the moment, however, we pause to examine more carefully the benefits provided by the current automation work.

The most obvious change in moving from the perspective afforded by the previous work to the one described here is the disappearance of the low-level language syntax. This cosmetic difference is of practical importance as it removes what often amounts to the first barrier of entry into a low-level program

proof endeavor, the requirement that analysis begin with a study of a program implemented in an unfamiliar, low-level language.

More importantly, myriad details of the machine which have no bearing on the nature of the function computed are absent from the characterization function's definition. In the case at hand, the only intricacy of the JVM that remains is the fixing operation described in the previous section. Even where such detail remains, notice that there is no reference to manipulation of machine state. In fact, our mechanization removes the entire device of the language interpreter, and with it, the need to reason about executions of a machine. This is perhaps a bigger win than might be evident, such reasoning being particularly cumbersome in the setting of the mechanized logic as it exposes the user to details related to configuring the rewriter to perform efficient symbolic simulation.

We emphasize here that though details of the machine and modeling device removed in the process of generating the characterization function are not relevant to the nature of the function computed by a program, they are relevant to *how* the program computes the function, and how ACL2 "reasons" about the program. Management of these details is the focus of the mechanization work described in the next section.

2.5 Our Approach

Our preliminary approach for mechanically generating verifiably correct program abstractions is based upon a process of composing symbolic simulation runs over fragments of a program's control flow graph. Roughly speaking, each symbolic simulation run over a fragment produces a simplified expression representing the effect on machine state of executing the fragment's associated low-level code. The simplified expression is "packaged up" into an ACL2 function definition and an ACL2 theorem asserting correspondence between executing the code and evaluating the newly-introduced function. Each composition step similarly produces a function/theorem pair characterizing a composite fragment, and the goal is to arrive at, through a sequence of such composition steps guided by control flow analysis, one function characterizing executions of the program as a whole, and one theorem certifying that the characterization is legitimate.

Our ability to produce such a function and theorem for a program of interest, and have ACL2 certification succeed without user involvement, hinges upon direct invocation of ACL2's internal simplifier in the generation process. Care is taken to ensure that the steps taken by the simplifier "off-line" on behalf of our mechanization are the steps taken when ACL2 certifies the theorem produced. We emphasize here that no changes to ACL2, itself, have been required. In particular, the ACL2 that certifies a correspondence theorem is the public-release version of the theorem prover, and as such provides the same soundness guarantee. In the remainder of this section we presents details of our preliminary approach, which we are working to complete for the language of the Java Virtual Machine, and to generalize for other languages amenable to formal operational modeling.

In the discussion that follows, we reference several concepts from compiler optimization theory. We give two definitions here, and several more, later, as the need arises. A *basic block* is a sequence of program instructions which control flow enters only at the beginning and exits only at the end. A program's *control flow graph (CFG)* is a directed graph whose nodes represent the program's basic blocks, and whose edges represent the program's transfers of control. A basic block's property of being free of internal branches out makes it straightforward to arrange for its symbolic simulation. Its property of being free of branches into the middle ensures that one simulation is enough to characterize that block's contribution to any computation involving its instructions. These properties enable us to characterize a program's CFG as a set of characterization function/correspondence theorem pairs, one pair for each of the CFG's basic blocks, in an initialization process we refer to as "block lifting."

To illustrate this process, consider the CFG of the addThree JVM method described in section 2.3, which consists of precisely one basic block representing the method's entire code array, repeated here for reference:

```
iinc 0 1
iinc 0 1
iinc 0 1
iload_0
ireturn
```

Chapter 3

Enhancements to ACL2

3.1 Introduction

In this chapter we describe some of the enhancements we have made to ACL2 itself. Here, we present a brief overview of the types of enhancements made and their motivations.

Minor improvements: Some of the improvements we have made are directed towards algorithmic efficiency and code maintainability. We present a representative sample of such changes in Section 3.2.

Meeting user expectations: In order to use ACL2 effectively, one must have a mental model of ACL2's operations. This mental model does not need to be particularly deep or detailed; but, if ACL2 behaves in ways apparently at odds with the user's model, he is likely to become frustrated and to perceive ACL2 as harder to use than need be.

Although ACL2 has been, for several years now, a mature product with an impressive track record of achievement, when we first started this project it still had a large number of "rough edges" and quirks. In Section 3.3, we describe our work to eliminate such troublesome behavior on the part of ACL2.

Command and control: In Section 3.4 we describe three new facilities designed to give the user greater control over certain details of ACL2's operations. These facilities were designed to overcome specific obstacles we encountered while trying to write an improved library of arithmetic lemmas. Thus, while the changes described in Section 3.3 were introduced to improve the ACL2 end-user's experience, the facilities described in this section are designed to improve the library writer's experience.

Improved context maintenance: Previously, when ACL2 was rewriting one of the branches of an `IF` expression, it did not always use all of the contextual information available to it. This problem was inherited from NQTHM

(a.k.a. the Boyer-Moore prover, the predecessor to ACL2). We have now fixed it. See Section 3.5.

Linear arithmetic and type-reasoning: Linear arithmetic and type-reasoning are two of the reasoning packages built into ACL2. Previously, the linear arithmetic package used type-reasoning at a low-level of its operations. Additionally, details of the linear arithmetic package's data representation were relied upon in several places outside the linear arithmetic package. These bits of tangled code caused us several difficulties. We discuss these issues and their solution in Section 3.6.

Non-linear arithmetic: ACL2 has had a linear arithmetic reasoning package since its inception. We have extended ACL2's arithmetic reasoning abilities by adding a non-linear arithmetic package. In Section 3.7 we describe our motivations and briefly describe the new package. We describe it more fully in Appendix I.

3.2 Some Minor Improvements

We have made several minor improvements to ACL2 which are, other than timing issues, generally invisible to the user. We present a representative sample of these changes here. The first, removing linear-alias rules, was prompted by a desire for a more easily maintainable code base. The other three were motivated by improved algorithmic efficiency. Their combination has resulted in linear arithmetic reasoning being sped up by as much as 90% on large problems. On small problems any changes in efficiency were undetectable.

3.2.1 Removing Linear-alias Rules

We eliminated the rule-class linear-alias. This rule class unnecessarily complicated ACL2's code base. The only use of the rule-class that we know of was in the nqthm books, which were an attempt to provide an embedding of the Nqthm logic and theorem prover into ACL2. But that facility was never used, as far as we know. So both linear-alias rules and the nqthm books have been eliminated.

3.2.2 Parent Trees

ACL2 uses a data structure known as a parent tree for certain heuristic purposes within the linear arithmetic procedures. Previously, the parent trees were scattered throughout a larger, more general, data structure. Under certain conditions merely accessing (as opposed to using) these parent trees could take up to 80% of the time used by linear arithmetic. We now pre-extract the parent trees and store them separately. Although the savings from this change are quite dramatic on occasion, in most cases there is no apparent difference either positive or negative.

3.2.3 Normalizing Polynomials

At the heart of the linear arithmetic decision procedure — described briefly in Section 3.7 and more fully in Appendix I — is the notion of "cancellation". For example, given the two inequalities

$$4 < -3 \cdot x + -7 \cdot a \tag{3.1}$$

$$3 < 2 \cdot x \tag{3.2}$$

we can cancel them by multiplying inequality (3.1) by 2 and inequality (3.2) by 3, and then adding the respective sides. Previously, this was almost exactly how cancellation was performed. However, we now normalize all inequalities before using them in the linear arithmetic decision procedure. That is, the above inequalities would now be stored as

$$4/3 < -1 \cdot x + 7/3 \cdot a$$

$$3/2 < x.$$

Note that by dividing each polynomial through by its leading coefficient, we do not have to do any cross-multiplication in order to cancel the two inequalities. Since the initial storing of an inequality is done but once, while any inequality may be cancelled with many others, the overall operations are much more efficient.

3.2.4 A Better Poly-memberp

Previously, when ACL2 had a new inequality it was considering adding to the linear arithmetic data-base, it would first determine whether an equal one was already present and, if so, the new one would be thrown away and ignored. Thus, if ACL2 already had the inequality $2 < x$ it would not try to add this exact inequality to the data-base a second time. However, it would still add the inequality $4 < 2 \cdot x$, even though this is "really" the same. By normalizing inequalities, ACL2 is able to catch such duplications.

Moreover, the above-mentioned normalization has aided in further improving this filtering so that a polynomial is ignored if an "obviously stronger" one is already present. We here provide some illustrative examples of the notion, obviously stronger:

1. The inequality $5 < x + y$ is stronger than $3 < x + y$.

2. The inequality $5 < x + y$ is stronger than $5 \leq x + y$.

3. We make no judgment about the relationship of $5 < x+y$ and $5 < x+2 \cdot y$. That is, if we write an inequality as const $<$ RHS, we only compare inequalities with equal RHSs.

3.3 Meeting Users Expectations

The changes described in this section were all motivated by complaints from users. The first two of these changes are in the released ACL2. The next one is ready to be moved into the main development branch — what will become ACL2 v2-8.

3.3.1 Regularizing Equality substitutions

Previously, within ACL2, equality substitutions were performed by several different functions. Unfortunately, the heuristics used to direct these substitutions differed somewhat from function to function. We have now regularized the situation. Since the details of this work are rather complicated and technical, we do not attempt to describe them in any detail here. Rather, we briefly describe one typical change.

When a goal being simplified contains a hypothesis of the form `(equal <rhs> <lhs>)` ACL2, under certain circumstances, performs the substitution suggested by the equality. Similarly, when ACL2 "sees" two inequalities of the form `(<= x y)` and `(<= y x)`, and both `x` and `y` are numeric, it can deduce that `x` and `y` are equal. This is done by the function `process-equational-polys`. Previously, ACL2 would perform the substitution suggested by this newly deduced equality within the function `process-equational-polys`. Unfortunately, the heuristics used to direct this substitution differed substantially from those used to control substitutions elsewhere in the ACL2 code base. We fixed this discrepancy by adding the deduced equality (`(equal x y)` in the example above.) explicitly to the clause, and allowing the "regular" substitution engine to do the work.

3.3.2 Eliminating Unwanted Hypothesis Elimination

Previous to v2-6, if a hypothesis of a goal could be established by contextual reasoning alone, it was removed from the goal. A simple example of the difficulties this can cause follows. We assume the use of ACL2 v2-5 and presence of an arithmetic library containing, at least,

```
(defthm comm-2-of-+
  (equal (+ x y z)
         (+ y x z)))
```

```
(defthm merge-consts-+
  (implies (and (syntaxp (consp c))
                (syntaxp (eq (car c) 'QUOTE))
                (syntaxp (consp d))
                (syntaxp (eq (car d) 'QUOTE)))
```

```
(equal (+ c d x)
       (+ (+ c d) x))))
```

Suppose we wanted to prove[1]

```
(defthm tricky
 (implies (integerp (+ -1/2 x))
          (integerp (+ 1/2 x))))
```

ACL2 does not know what to do with this. But, it does know that if `y` is an integer, than so is `(+ 1 y)`. We know that `(+ 1/2 x)` is "really" the same as `(+ 1 (+ -1/2 x))` and that `(+ -1/2 x)` is an integer; therefore the lemma should be obvious. Perhaps if we drew ACL2's attention to these facts, ACL2 would also find the lemma obvious. We therefore try the hint

```
(thm
 (implies (integerp (+ -1/2 x))
          (integerp (+ 1/2 x)))
 :hints (("Goal" :use ((:instance
                        (:theorem
                         (implies (integerp y)
                                  (integerp (+ 1 y))))
                         (y (+ -1/2 x)))))))
```

But, when this is given to ACL2, it claims that the hint is so obvious it should be removed — and then fails because it does not know the fact that it just removed! Here are the relevant parts from a transcript of the failing proof attempt:

```
Goal''
(IMPLIES (AND (OR (NOT (INTEGERP (+ -1/2 X)))
                  (INTEGERP (+ 1 -1/2 X)))
              (INTEGERP (+ -1/2 X)))
         (INTEGERP (+ 1/2 X))).


This simplifies, using primitive type reasoning, to
```

[1]The need for such a lemma can arise when one is proving something about even numbers. By an odd coincidence one of the authors of this report was asked twice within the span of a month how to prove this very theorem.

```
Goal'''
(IMPLIES (INTEGERP (+ -1/2 X))
         (INTEGERP (+ 1/2 X))).


******** FAILED ********  See :DOC failure  ******** FAILED ********
```

Since the hint was removed using primitive type reasoning, there is little that the user can do to prevent this. This is wrong.

Here are the results in v2-6:

```
Goal''
(IMPLIES (AND (IMPLIES (INTEGERP (+ -1/2 X))
                       (INTEGERP (+ 1 -1/2 X)))
              (INTEGERP (+ -1/2 X)))
         (INTEGERP (+ 1/2 X))).


But simplification reduces this to T, using the :definition SYNTAXP,
the :executable-counterparts of BINARY-+, CAR, CONSP and EQ, primitive
type reasoning and the :rewrite rule MERGE-CONSTS-+.

Q.E.D.
```

As can be seen, type reasoning is no longer sufficient to remove a hypothesis from a goal. While it was not common to encounter this problem before, it was quite frustrating and difficult to overcome when one did.

3.3.3 Forward Chaining and Linear Arithmetic

At present the conclusions of forward chaining rules are not used by the arithmetic reasoning facilities in ACL2. We have an experimental extension to the forward chaining mechanism which rectifies this.

Before presenting an example of the benefits to be derived from this change, we briefly describe the use of forward-chaining rules.

Suppose we wish to reason about sets, and that we will represent them as lists. We might then define

```
(defun setp (x)
  (true-listp x)).
```

If we now leave this definition enabled, rules about `setp` — that is, rules which mention `setp` in the right hand side of their conclusion — will not get used. That is, the conclusion of a rule such as

```
(defthm setp-set-union
  (implies (and (setp x)
                (setp y))
           (setp (set-union x y)))).
```

will never match with anything — and the rule will therefore not be used — because a potential matching term such as `(setp (set-union a b))` will be immediately rewritten to `(true-listp (set-union a b))` using the definition of `setp` before the rule `setp-set-union` is seen.

However, if we just disable it, ACL2 will be unable to prove that anything is a `setp`.

This is what forward chaining rules were designed for. If we admitted the rules

```
(defthm implies-setp
  (implies (true-listp x)
           (setp x)))

(defthm setp-fwd
  (implies (setp x)
           (true-listp x)))
```

and then disabled the definition `setp`, ACL2 will be able to both use rules about `setp` and to prove when something is a `setp`.

The first rule, `implies-setp`, tells ACL2 how to prove that something is a `setp`. Normally, this rule would loop with either the definition or the second rule, `setp-fwd`, but the definition was disabled and `setp-fwd` is a forward-chaining rule. Forward-chaining rules are only used before rewriting begins — they augment the context in which rewriting takes place. They do not interact directly with rewrite rules, and so no looping can take place. In this example, `setp-fwd` will cause ACL2 to take note of the fact that `x` is a `true-listp` whenever `x` is known to be a `setp`. Therefore, for instance, given the definition and lemma

```
(defun set-union (x y)
  (append x y))

(defthm true-listp-append
  (implies (and (true-listp x)
                (true-listp y))
           (true-listp (append x y))))
```

ACL2 would be able to prove

```
(defthm setp-set-union
  (implies (and (setp x)
                (setp y))
           (setp (set-union x y)))).
```

It would do so by first noting that `x` and `y` are both `setp` and using `setp-fwd` twice to derive the facts that `x` and `y` are also `true-listp`. ACL2 would then be able to use the rewrite rules `implies-setp`, `true-listp-append`, and the definition `set-union` to prove the theorem by rewriting.

Given the above description of how forward-chaining is used, one might hope that one could treat predicates like

```
(defun registerp (x)
   (and (integerp x)
        (<= -128 x)
        (< x 128)))
```

similarly. However, this was not previously the case. Given the seemingly analogous lemmas,

```
(defthm implies-registerp
  (implies (and (integerp n)
                (<= -128 n)
                (< n 128))
           (registerp n)))

(defthm registerp-fwd
  (implies (registerp n)
           (and (integerp n)
                (<= -128 n)
                (< n 128)))
  :rule-classes :forward-chaining)
```

One would hope that one could easily prove the following theorem.

```
(defthm fails
 (implies (and (registerp n)
               (registerp (+ 7 n)))
          (registerp (+ 3 n))))
```

Unfortunately, this was not the case. Note that above we said that forward-chaining rules augment the context in which rewriting takes place, and that the lemma `setp-set-union` was proven via rewriting. The lemma `fails` however requires linear arithmetic reasoning to be proved. However, the relevant facts were previously stored only in the type-alist and not in the arithmetic database where they could be effectively used.

With our experimental extension to the forward chaining mechanism, the conclusions of forward-chaining rules are available to linear arithmetic, and the above theorem is proven automatically..

3.4 Command and Control

The changes described in this section were motivated by a desire to write rules which had no natural expression in ACL2 at the time. They all add facilities to control the operations of ACL2 more precisely.

The first of these allows limits to be placed on the resources used to relieve a hypothesis. The other two provide facilities to guide ACL2's use of rewrite rules in a more intelligent and user-controllable manner.

3.4.1 Backchain Limits

The released ACL2 has three main mechanisms which it uses when trying to relieve a hypothesis — type reasoning, linear arithmetic, and backchaining. Type reasoning implements the strategy of looking up a hypothesis in the current context (it also implements the strategy 'this is obvious' in the sense that it is obvious that `(+ x y)` is a number). Only the other two strategies typically take up any significant resources, and it is these whose use may be restricted with backchain limits.

When, during a call to `rewrite` within the ACL2 simplifier, one of the hypotheses of a rewrite rule cannot be established by type reasoning alone, ACL2 adds its negation to the list `ancestors` and recursively calls `rewrite` on the hypothesis. This is known as backchaining. If the hypothesis involves an equality or inequality, `rewrite` will call the linear arithmetic package. Thus, by making sure that `ancestors` does not grow 'too long', we can limit the amount of space and time the system will expend while trying to relieve the hypotheses of a rule.

The user can now specify when `ancestors` is too long on a global, per-rule, or per-hypothesis basis. Previously, there had been a global limit on the length of `ancestors`, but it was compiled in. Now a user can `(set-backchain-limit <value>)`, where value can be either a non-negative number or `NIL` meaning infinity, and ACL2 will not backchain more than `<value>` times. More useful however are the other, local, methods of setting backchain limits.

There is now a `:backchain-limit` field attached to rewrite and linear rules which the user may set when defining a rule. (These are the only two rule-classes, besides meta rules, which use backchaining.) The legal values of this field are

1. a non-negative number or `NIL`
2. a list, each element of which is either a non-negative integer or `NIL`, and whose length is equal to the number of hypotheses of the rule.

For example:

```
(defthm ex1
;; ACL2 will not backchain more than twice when attempting to
;; establish any of the hypotheses.
```

```
  (implies (and (pred1 x)
                (pred2 y)
                (pred3 x z))
           (concl x y z))
  :rule-classes ((:rewrite :backchain-limit 2)))

(defthm ex2
;; No additional restrictions are imposed on how ACL2 establishes
;; the first hypothesis.  Type reasoning alone is allowed for the
;; second rule. ACL2 will not backchain more than twice when
;; attempting to establish the third hypothesis.
  (implies (and (pred1 x)
                (pred2 y)
                (pred3 x z))
           (concl x y z))
  :rule-classes ((:rewrite :backchain-limit (nil 0 2))))
```

Note that backchain limits act negatively in that they can only eliminate a possible course of action, not suggest one. Note also that while the global backchain limit is absolute, the local backchain limits are relative to the current situation. When ACL2 first enters the simplifier the current backchain limit is set to the global value. The local values can only decrease the current backchain limit. More specifically, when ACL2 is about to try to establish a hypothesis, it calculates a new current backchain limit using

```
(defun new-backchain-limit (new-offset old-limit ancestors)

;; New-offset is the backchain limit for the current hypothesis,
;; old-limit is the old backchain limit, and (length ancestors)
;; is how many times we have backchained so far.  Recall that a
;; null limit is equivalent to an infinite one.

  (cond ((null new-offset)
         old-limit)
        ((null old-limit)
         (+ (length ancestors) new-offset))
        (t
         (min (+ (length ancestors) new-offset)
                 old-limit)))),
```

and it is this limit which will be compared to the length of `ancestors`.

We imagine that one common use of backchain limits will be to allow only type reasoning for the relief of selected hypotheses. This can be done by setting the backchain limit to zero for those rules, and would allow an otherwise potentially expensive rule to fire only when it is 'obviously' true. We have used it extensively to control the expense of certain linear rules about floor and mod. For these rules, we have used a backchain limit of one. We have found this to

be a good compromise which allows ACL2 to put a small amount of effort into proving the rules, but only a small amount. A backchain limit of zero would make the rules too weak, and thereby miss too many chances of being used; a larger backchain limit would make this collection of rules too expensive for common use.

3.4.2 Meta Rules

Meta rules can be used to extend the ACL2 simplifier with hand-written code which transforms ACL2 terms to equivalent ones. For instance, one could write a function, say `cancel-common-addends` which takes an equality between two sums and cancels the common addends from both sides.

```
(defun cancel-common-addends (term)
  (if (equality-between-sums term)
      (cancel-common-addends-helper term)
    term))
```

One could then prove that the ACL2 term `term` is equal to `(cancel-common-addends term)` and submit the theorem

```
(defthm cancel-common-addends-meta-thm
  (equal (eva term alist)
         (eva (cancel-common-addends term) alist))
  :rule-class ((:meta :trigger-fns (equal)))).
```

This theorem states that given an ACL2 term, `term`, and any possible set of bindings for the variables of `term`, evaluating `term` under those bindings is equal to evaluating the result of `(cancel-common-addends term)` under the same bindings. (the variable bindings are represented by the use of `alist` above.) After admitting `cancel-common-addends-meta-thm`, each time ACL2 encountered an equality, it would apply `cancel-common-addends` to the equality and, if the result differed (syntactically) from the original equality, use the result in place of the original equality.

This mechanism allowed the construction of rules which could not be expressed using rewrite rules alone, such as a rule which would cancel 'like' terms from both sides of arbitrary arithmetic equalities no matter where within the equated expressions these terms were to be found.

However, it was not previously possible to write a meta rule which would cancel like factors from both sides of arbitrary arithmetic inequalities exactly when doing so would not induce a case-split on the sign of the canceled factors. The metafunction must operate blind to the context in which it is called. In order to eliminate this difficulty, we developed a new, augmented, type of metafunction — the *extended* metafunction. Just as meta rules generalize regular rewrite rules, the new extended meta rules generalize meta rules in that they can gather information and test hypotheses *before* deciding how to proceed rather than afterwards.

Traditionally, metafunctions have taken just one argument, the term being rewritten. The new extended metafunctions take two additional arguments which, in effect, pass the current context in to the metafunction. A typical extended meta rule would look something like:

```
(equiv (eva x a)
       (eva (meta-fn x mfc state) a))),
```

where `state` is literally the symbol `state`. These new arguments empower the user to call certain theorem proving functions, like `type-set` and `rewrite`, from within a meta function as well as to access information about the context of the metafunction call, like the `type-alist` and the `current-clause`.

The function `(mfc-clause mfc)` returns the (translation of the) clause containing the term `x` presently being rewritten by the metafunction, and `(mfc-type-alist mfc)` returns the type-alist governing the occurrence of `x`. Using these, a metafunction can tell what the current goal is and what is trivially known at the moment. One simple use for this would be to write rules which use their own criteria for the selection of bindings for free variables. Similarly, `(mfc-ts term mfc state)` returns the type-set of `term`, while `(mfc-rw term obj iffp mfc state)` returns the result of rewriting `term` with objective `obj` and the equivalence relation described by `iffp`[2]. Note particularly that `term` can be any ACL2 term — not just the term presently being rewritten. These latter two functions allow a metafunction to find out what ACL2 'knows', and so to customize its behavior to the current context.

During the execution of a metafunction by the theorem prover, the expressions above compute the results specified. However, there are no axioms about `mfc-ts` and `mfc-rw`: they are uninterpreted function symbols. Thus, during the proof of the correctness of a metafunction, no information is available about the results of these functions. *They can be used for heuristic purposes only.* A metafunction may use these functions to decide whether to perform a given transformation, but the transformation must be sound regardless.

We now present a couple of pedagogical examples of the use of extended metafunctions in the form of a script for an ACL2 session. We present some further, more practical, examples in Appendix C. We also present a larger example in Appendix D. This last mentioned example is taken from our library of arithmetic lemmas and has proven to be very useful.

```
; I will define (foo i j) simply to be (+ i j).
; But I will keep its definition disabled so the theorem prover
; doesn't know that.  Then I will define an extended metafunction
; that reduces (foo i (- i)) to 0 provided i has integer type and the
; expression (< 10 i) occurs as a hypothesis in the clause.
```

[2] `Obj` must be `t`, `nil`, or `?`, and describes the objective of the rewriter: try to show that `term` is true, false, or anything respectively. `Iffp` is either `nil` or non-`nil`. The former means return a term that is equal to `term`. The latter means return a term that is propositionally equivalent to `term`.

```
; Note that (foo i (- i)) is 0 in any case.


(defun foo (i j) (+ i j))


(defevaluator eva eva-lst ((foo i j)
                           (unary-- i)

; I won't need these two cases until the last example below, but I
; include them now.
                           (if x y z)
                           (integerp x)))


(set-state-ok t)


(defun metafn (x mfc state)
  (cond
   ((and (consp x)
         (equal (car x) 'foo)
         (equal (caddr x) (list 'unary-- (cadr x))))


; So x is of the form (foo i (- i)).  Now I want to check some other
; conditions.


    (cond ((and (ts-subsetp (mfc-ts (cadr x) mfc state)
                            *ts-integer*)
                (member-equal `(NOT (< '10 ,(cadr x)))
                              (mfc-clause mfc)))
           ''0)
          (t x)))
   (t x)))


(defthm metafn-correct
  (equal (eva x a) (eva (metafn x mfc state) a))
  :rule-classes ((:meta :trigger-fns (foo))))


(in-theory (disable foo))
```

```
; The following will fail because the metafunction won't fire.
; We don't know enough about i.


(thm (equal (foo i (- i)) 0))


; Same here.


(thm (implies (and (integerp i) (< 0 i)) (equal (foo i (- i)) 0)))


; But this will work.


(thm (implies (and (integerp i) (< 10 i))
              (equal (foo i (- i)) 0)))


; This won't, because the metafunction looks for (< 10 i) literally,
; not just something that implies it.

(thm (implies (and (integerp i) (< 11 i))
              (equal (foo i (- i)) 0)))


; Now I will undo the defun of metafn and repeat the exercise, but
; this time check the weaker condition that (< 10 i) is provable
; (by rewriting it) rather than explicitly present.


(ubt 'metafn)

(defun metafn (x mfc state)
  (cond
   ((and (consp x)
         (equal (car x) 'foo)
         (equal (caddr x) (list 'unary-- (cadr x))))
    (cond ((and (ts-subsetp (mfc-ts (cadr x) mfc state)
                            *ts-integer*)
                (equal (mfc-rw `(< '10 ,(cadr x)) t t mfc state)
                       *t*))

; The mfc-rw above rewrites (< 10 i) with objective t and iffp t.  The
; objective means the theorem prover will try to establish it.  The
```

```
; iffp means the theorem prover can rewrite maintaining propositional
; equivalence instead of strict equality.


          ''0)
         (t x)))
   (t x)))


(defthm metafn-correct
  (equal (eva x a) (eva (metafn x mfc state) a))
  :rule-classes ((:meta :trigger-fns (foo))))


(in-theory (disable foo))


; Now it will prove both:


(thm (implies (and (integerp i) (< 10 i))
              (equal (foo i (- i)) 0)))


(thm (implies (and (integerp i) (< 11 i))
              (equal (foo i (- i)) 0)))


; Now I undo the defun of metafn and change the problem entirely.
; This time I will rewrite (integerp (foo i j)) to t.  Note that
; this is true if i and j are integers.  I can check this
; internally, but have to generate a hyp-fn to make it official.


(ubt 'metafn)

(defun metafn (x mfc state)
  (cond
   ((and (consp x)
         (equal (car x) 'integerp)
         (consp (cadr x))
         (equal (car (cadr x)) 'foo))

; So x is (integerp (foo i j)).  Now check that i and j are
; ``probably'' integers.
```

```
    (cond ((and (ts-subsetp (mfc-ts (cadr (cadr x)) mfc state)
                            *ts-integer*)
                (ts-subsetp (mfc-ts (caddr (cadr x)) mfc state)
                            *ts-integer*))
           *t*)
          (t x)))
   (t x)))

; To justify this transformation, I generate the appropriate hyps.


(defun hyp-fn (x mfc state)


  (declare (ignore mfc state))


; The hyp-fn is run only if the metafn produces an answer different
; from its input.  Thus, we know at execution time that x is of the
; form (integerp (foo i j)) and we know that metafn rewrote
; (integerp i) and (integerp j) both to t.  So we just produce their
; conjunction.  Note that we must produce a translated term; we
; cannot use the macro AND and must quote constants!  Sometimes you
; must do tests in the hyp-fn to figure out which case the metafn
; produced, but not in this example.

          `(if (integerp ,(cadr (cadr x)))
               (integerp ,(caddr (cadr x)))
               'nil))

(defthm metafn-correct
  (implies (eva (hyp-fn x mfc state) a)
           (equal (eva x a) (eva (metafn x mfc state) a)))
  :rule-classes ((:meta :trigger-fns (integerp))))


(in-theory (disable foo))


; This will not be proved.


(thm (implies (and (rationalp x) (integerp i)) (integerp (foo i j))))
```

```
; But this will be.


(thm (implies (and (rationalp x)
                   (integerp i)
                   (integerp j))
              (integerp (foo i j))))
```

3.4.3 Bind-free

The bind-free facility was designed to solve many of the same types of problems as meta rules, but to be much more "light weight". Meta rules can be difficult to reason about, and are intimidating to most users of ACL2 — even the most sophisticated. Mike Smith of AMD (Advanced Micro Designs) pointed out that much of the power of meta rules could be gotten if one could choose bindings for free variables under programatic control. We then fleshed out his suggestion and implemented bind-free rules.

A bind-free rule is one which uses a bind-free hypothesis. An example rule is the following

```
(defthm cancel-matching-addends-thm
  (implies (and (rationalp lhs)
                (rationalp rhs)
                (bind-free (find-match-in-plus-nests lhs rhs) (x)))
           (equal (equal lhs rhs)
                  (equal (+ (- x) lhs) (+ (- x) rhs)))))
```

The hypothesis `(bind-free (find-match-in-plus-nests lhs rhs) (x))` is used to bind the free variable `x` which appears in the right-hand side above. When `cancel-matching-addends-thm` is used, `x` would presumable be bound to a common addend which appeared in both `lhs` and `rhs`; other rules would then simplify the new terms `(+ (- x) lhs)` and `(+ (- x) rhs)`. However, `bind-free` is logicly defined to always return `T`. Thus, proving `cancel-matching-addends-thm` is logicly equivalent to proving

```
(defthm example
  (implies (and (rationalp lhs)
                (rationalp rhs))
           (equal (equal lhs rhs)
                  (equal (+ (- x) lhs) (+ (- x) rhs)))))
```

in which `x` truly is a free variable.

Note that a bind-free hypothesis does not, in general, deal with the meaning or semantics or values of the terms, but rather with their syntactic forms. Before attempting to write a rule which uses bind-free, the user should be familiar with syntaxp and the internal form that ACL2 uses for terms. This internal form is

similar to what the user sees, but there are subtle and important differences. Trans can be used to view this internal form.

We begin our description of bind-free by examining the example above in some detail.

We wish to write a rule which will cancel "like" addends from both sides of an equality. Clearly, one could write a series of rules such as

```
(defthm the-hard-way-2-1
   (equal (equal (+ a x b)
                 (+ x c))
          (equal (+ a b)
                 (fix c))))
```

with one rule for each combination of positions the matching addends might be found in (if one knew before-hand the maximum number of addends that would appear in a sum). But there is a better way. (In what follows, we assume the presence of an appropriate set of rules for simplifying sums.)

Consider the following definitions and theorem:

```
(defun intersection-equal (x y)
  (cond ((endp x)
         nil)
        ((member-equal (car x) y)
         (cons (car x) (intersection-equal (cdr x) y)))
        (t
         (intersection-equal (cdr x) y))))

(defun plus-leaves (term)
  (if (eq (fn-symb term) 'binary-+)
      (cons (fargn term 1)
            (plus-leaves (fargn term 2)))
    (list term)))

(defun find-match-in-plus-nests (lhs rhs)
  (if (and (eq (fn-symb lhs) 'binary-+)
           (eq (fn-symb rhs) 'binary-+))
      (let ((common-addends (intersection-equal (plus-leaves lhs)
                                                (plus-leaves rhs))))
        (if common-addends
            (list (cons 'x (car common-addends)))
          nil))
    nil))
```

And now for our bind-free rule.

```
(defthm cancel-matching-addends-equal
  (implies (and (rationalp lhs)
                (rationalp rhs)
```

```
        (bind-free (find-match-in-plus-nests lhs rhs) (x)))
   (equal (equal lhs rhs)
          (equal (+ (- x) lhs) (+ (- x) rhs)))))
```

Let us examine how this rule is applied to the following term.

```
(equal (+ 3 (expt a n) (foo a c))
       (+ (bar b) (expt a n)))
```

As mentioned above, the internal form of an ACL2 term is not always what one sees printed out by ACL2. In this case, by using :trans one can see that the term is stored internally as

```
(equal (binary-+ '3
                 (binary-+ (expt a n) (foo a c)))
       (binary-+ (bar b) (expt a n))).
```

When ACL2 attempts to apply `cancel-matching-addends-equal` to the term under discussion, it first forms a substitution that instantiates the left-hand side of the conclusion so that it is identical to the target term. This substitution is kept track of by the substitution alist:

```
((lhs . (binary-+ '3
                  (binary-+ (expt a n) (foo a c))))
 (rhs . (binary-+ (bar b) (expt a n)))).
```

ACL2 then attempts to relieve the hypotheses in the order they were given. Ordinarily this means that we instantiate each hypothesis with our substitution and then attempt to rewrite the resulting instance to true. Thus, in order to relieve the first hypothesis, we rewrite:

```
(rationalp (binary-+ '3
                     (binary-+ (expt a n) (foo a c)))).
```

Let us assume that the first two hypotheses rewrite to `T`. How do we relieve the bind-free hypothesis? Just as for a syntaxp hypothesis, ACL2 evaluates `(find-match-in-plus-nests lhs rhs)` in an environment where `lhs` and `rhs` are instantiated as determined by the substitution. In this case we evaluate

```
(find-match-in-plus-nests '(binary-+ '3
                                     (binary-+ (expt a n) (foo a c)))
                          '(binary-+ (bar b) (expt a n))).
```

Observe that, just as in the case of a syntaxp hypothesis, we substitute the quotation of the variables bindings into the term to be evaluated. The result of this evaluation, `((X . (EXPT A N)))`, is then used to extend the substitution alist:

```
((x . (expt a n))
 (lhs . (binary-+ '3
                  (binary-+ (expt a n) (foo a c))))
 (rhs . (binary-+ (bar b) (expt a n)))),
```

and this extended substitution determines `cancel-matching-addends-equal`'s result:

```
(equal (+ (- x) lhs) (+ (- x) rhs))
==>
(equal (+ (- (expt a n)) 3 (expt a n) (foo a c))
       (+ (- (expt a n)) (bar b) (expt a n))).
```

When this rule fires, it adds the negation of a common term to both sides of the equality by selecting a binding for the otherwise-free variable `x`, under progammatic control. Note that other mechanisms such as the binding of free-variables may also extend the substitution alist.

An extended bind-free hypothesis is similar to the simple type described above, but it uses two additional variables,`mfc` and `state`. These two variables give access to the functions `mfc-xxx`; see the discussion of extended-metafunctions in the previous section. As described there, `mfc` is bound to the so-called `metafunction-context` and `state` to ACL2's `state`. See Appendix E for examples of the use of these extended bind-free hypotheses.

3.5 Improved Context Maintenance

Normally, when ACL2 rewrites one of the branches of an `IF` expression, it "knows" the truth or falsity (as appropriate) of the test. More precisely, if `x` is a term of the form `(IF <test> <true-branch> <false-branch>)`, when ACL2 rewrites `<true-branch>`, it "knows" that `<test>` is true , and when it rewrites `<false-branch>` it "knows" that `<test>` is false. This is certainly what one would expect.

However, previously, if test was such that it would print as `(AND <test1> <test2>)` – so that `x` would print as `(IF (AND <test1> <test2>) <true-branch> <false-branch>)` – when ACL2 rewrote `<true-branch>` it only knew that `(AND <test1> <test2>)` was true — it did not know that `<test1>` was true nor that `<test2>` was true, but only that their conjunction was true. There were also other situations in which ACL2's reasoning was weak about the test of an `IF` expression. This behavior was inherited from NQTHM (aka, the Boyer-Moore Theorem Prover).

We have fixed the problem in a comprehensive manner. Now, whenever ACL2 is rewriting one of the branches of an `IF` expression, it knows what one would expect about `<test>`. This new facility took several iterations to get right — that is to (1.) cover all the possible combinations and (2.) not be too computationally expensive. Even after that was done, other difficulties arose. These other difficulties are rather technical in nature, so we relegate their description and solution to Appendix G.

3.6 Linear Arithmetic and Type Reasoning

3.6.1 Introduction

In this section, we discuss our work with certain low-level aspects of linear arithmetic and type reasoning. Before we begin, however, we provide a little terminology. In this section, linear arithmetic will refer to the central linear arithmetic decision procedure; this consists, crudely, in taking the transitive closure of the inequalities under "cancellation" as described briefly Section 3.2.3 and more fully in I.2. Note in particular that this does not include the use of linear lemmas. We shall refer to the use of both linear arithmetic and linear lemmas as simply the arithmetic procedures. Type reasoning we do not attempt to describe here except to say that it can, in several ways, be thought of as the "fundamental" reasoning method in ACL2. It is not unreasonable to think of it as answering the question, "What do I know?" and the other procedures as more oriented towards answering "What can I derive from what I know?"

Previously, linear arithmetic had been integrated into the ACL2 rewriter through the use of linear lemmas. This integration has proved very powerful. Linear arithmetic was also integrated with type reasoning, but at a very low level. For the curious, we briefly describe this integration below, but first we wish to mention that the details are not important; only their consequences.

Linear arithmetic had been well integrated with parts of ACL2, but ACL2 had not been fully integrated with linear arithmetic. That is, the boundaries between the linear arithmetic algorithm and the rest of ACL2 were very unclear. Intimate details of the internal representation of linear arithmetic data structures were relied upon in several places in the code. Linear arithmetic used type reasoning at too low a level. All of this made both code maintenance and experimentation difficult — heuristics and actions were too intimately intertwined. It also made it impossible to layer another procedure such as nonlinear arithmetic on top the existing arithmetic procedures.

We now briefly describe the old integration of linear arithmetic with type reasoning. We do so by example. Let us assume that ACL2 had just derived the new inequality, $4 < y + x$. ACL2 would immediately use type reasoning to determine if y was negative — if it was, ACL2 could assume that $4 < x$ was also a valid inequality. It did not, however, store the inequality $y < 0$ anywhere. This role is now played by the function `polys-from-type-set` described in Section I.3.1 as "creating polys from `type-set`."

3.6.2 Disentangling and Reintegrating

In imitation of type-set-a and type-set-b, we created linear-a and linear-b. Linear-a does not use type-set, and so type-set can be made to use it without creating a horrid mutual-recursion mess. Linear-b does use type-set and reproduces the earlier interface. The result is as described in I.4.

3.6.3 Benefits Accrued

- The code is clearer and easier to maintain or modify.
- We were able to write and use `polys-from-type-set` for use by the nonlinear arithmetic procedure. Without this, nonlinear arithmetic would have been severely weakened.
- Type-set can use linear arithmetic. We have an experimental extension to type-set, which allows it to use linear arithmetic. While not a dramatic improvement, it allows type-set to decide more problems than it had before. This will get incorporated into the main development branch shortly.
- We have an experimental extension to forward-chaining which will allow the conclusions of forward-chaining rules to be used by linear arithmetic. See Section 3.3.3.
- Heuristic use of add-polys by meta or bind-free rules We have found it handy to be able to query the arithmetic database in a quick and clean manner for the purpose of controlling certain meta and bind-free rules. See ...in the arithmetic-4/....

3.7 Nonlinear Arithmetic

Our addition of a nonlinear arithmetic reasoning package to ACL2 has greatly increased the theorem prover's ability to automatically prove a large class of lemmas. In this section we present a brief overview of the new facility and describe its underlying logic. This material is taken from the introduction to Appendix I, where we describe the nonlinear arithmetic package more thoroughly. We also present an example of the type of lemma which motivated our work.

3.7.1 Motivational example

We present a challenge problem that Matt Kauffman, one of the authors of ACL2, sent to the ACL2 mailing list a couple of years ago. While he was able to prove it, he used eleven helper lemmas to do so and was hoping someone had a library that would have made the job easier.

```
(defthm arith-lemma-3
  (implies (and (rationalp a) (rationalp b) (rationalp c)
                (rationalp d) (rationalp e) (rationalp f)
                (rationalp g) (rationalp h) (rationalp i)
                (<= a d)
                (<= (+ (- (* h a))
                       (* f a))
```

```
            (+ (- (* h c))
               (* g c)
               b))
         (< i h)
         (< h g)
         (< e a)
         (<= g f))
   (<= (+ (- (* i e))
          (* e f))
       (+ (- (* i d))
          (* h d)
          (- (* h c))
          (* g c)
          b))))
```

During our investigations we realized that his eleven lemmatta could be replaced with just one linear lemma. Unfortunately, this lemma is very sensitive to seemingly irrelevant details such as the addition of extra hypotheses and the naming of the variables. Our irritation with this state of affairs led us to seek a better solution and resulted in the first version of our non-linear extensions to the linear arithmetic package. Since then, this has been one of the standard test lemmas we use whenever we modify the non-linear extensions.

3.7.2 Theory

We briefly describe here the theory behind the linear and nonlinear arithmetic procedures and provide a couple of examples of their use. Inequalities can be combined by cross-multiplication and addition to permit the deduction of an additional inequality. For example, if $0 < poly1$ and $0 < poly2$, and c and d are positive rational constants, then $0 < c \cdot poly1 + d \cdot poly2$. Here, we are using the facts that multiplication by a positive rational constant does not change the sign of a polynomial and that the sum of two positive polynomials is itself positive. This is linear arithmetic. Also, given the above, we have $0 < c \cdot poly1 \cdot poly2$. In this nonlinear case, we are using the fact that the product of two positive polynomials is itself positive.

3.7.3 Linear Example

Now suppose we want to prove

$$3 \cdot x + 7 \cdot a < 4 \quad \wedge \quad 3 < 2 \cdot x \quad \Longrightarrow \quad a < 0.$$

We can do this by assuming the two hypotheses and the negation of the conclusion, and looking for a contradiction. We therefore start with the three

inequalities:

$$0 < -3 \cdot x + -7 \cdot a + 4 \tag{3.3}$$

$$0 < 2 \cdot x + -3 \tag{3.4}$$

$$0 \leq a. \tag{3.5}$$

We cross-multiply and add the first two — that is, multiply inequality (3.3) by two and inequality (3.4) by three, and then add the respective sides. This yields

$$0 < -14 \cdot a + -1. \tag{3.6}$$

Note that the new inequality does not mention x. If we choose two inequalities with the same leading term and leading coefficients of opposite sign, we can generate an inequality in which that leading term is not present. This is the general strategy employed by the linear arithmetic decision procedure.

If we next cross-multiply and add inequality (3.6) with inequality (3.5), we get

$$0 < -1, \tag{3.7}$$

a false polynomial. We have, therefore, proved our theorem.

This process illustrated above of cross-multiplying and adding two inequalities will be referred to as "cancelling" the two inequalities. We shall refer to such obviously false inequalities as (3.7) as "contradictions," and speak of any process that results in one of these as "generating a contradiction."

3.7.4 Non-linear Example

Next, suppose that we have the three assumptions

$$3 \cdot x \cdot y + 7 \cdot a < 4 \quad \text{or} \quad 0 < -3 \cdot x \cdot y + -7 \cdot a + 4 \tag{3.8}$$

$$3 < 2 \cdot x \quad \text{or} \quad 0 < 2 \cdot x + -3 \tag{3.9}$$

$$1 < y \quad \text{or} \quad 0 < y + -1, \tag{3.10}$$

and we wish to prove that $a < 0$. We proceed by assuming the negation of our goal, $0 <= a$, and looking for a contradiction.

Note that in this case no two inequalities above have a leading term in common. In this situation there are no cancellations which can be performed. However, (3.8) has a product as its leading term, $x \cdot y$, and for each of the factors of that product, x and y, there is an inequality which has such a factor as a leading term. When nonlinear arithmetic is enabled, ACL2 will multiply[3] (3.9) and (3.10), obtaining

$$0 < 2 \cdot x \cdot y + -3 \cdot y + -2 \cdot x + 3. \tag{3.11}$$

The addition of this polynomial will allow cancellation to continue[4] and, in this case, we will prove our goal. Thus, just as ACL2 adds two polynomials

[3] How to multiply two inequalities will be described below.

[4] Inequality 3.11 can be canceled with 3.8. The result can be canceled with 3.10, and so on. The final cancellation will be with the negation of our goal.

together when they have the same largest unknown of opposite signs in order to create a new smaller polynomial, ACL2 can now multiply polynomials together when the product of their largest unknowns is itself the largest unknown of another polynomial.

Chapter 4

Library Development

4.1 Introduction

We have developed an improved library of lemmas about arithmetic functions. Some of this work involved carefully examining existing libraries to determine their strengths or weaknesses. However, a majority of our time was spent experimenting with "what if" scenarios. As mentioned above, we were often frustrated with the expressiveness of ACL2 rule-classes. We went through many iterations of adding experimental facilities to ACL2, writing rules using these new facilities, and debugging our initial ideas. (See the Sections on bind-free rules, Section 3.4.3, and Extended Meta rules, Section 3.4.2, for a description of two new rule classes developed.) The result of all this is a library which allows much simpler proofs of arithmetic lemmas than existed before. As above, see Appendices A and B for some example lemmas which we can now prove more easily.

4.2 Library Design

Our approach to library design has, so far, been a fairly ad-hoc process. At its lowest level, our library has a set of simple rules to impose an order of functional nesting. A typical rule is

```
(defthm |(- (+ x y))|
  (equal (- (+ x y))
         (+ (- x) (- y)))).
```

This rule makes sure that - only appears inside a +, never outside it. Our order of functional nesting is +, -, *, /. Above this, we have rules which gather "like terms" or cancel them from both sides of an equality or inequality. Two examples of their results are:

```
(+ x (* 3 y) (* 2 x))  ===> (+ (* 3 x) (* 3 y))
```

```
(< (+ w x (* 2 y))     ===>  (< (+ w x)
   (+ u (* 3 y)))              (+ u y)).
```

These latter rules have generally been chosen using a pre-calculus notion of "simpler," but have been supplemented by examining failed proofs and looking for missed simplifications. Our development of the extended meta rules and the bind-free rules was driven by our desires to better specify some of these rules.

Above these rules we have built a book of rules about `floor` and `mod`. This book is still somewhat incomplete and disorganized — we have not yet taken properly taken advantage of bind-free rules to generalize some of the current simplification rules.

4.3 Conclusion

The library that we have developed is much better than what came before. As above, see Appendices A and B for some example lemmas which we can now prove more easily. However, we still have a long way to go. With the modifications to ACL2 described in this report, we believe that we now have the tools in hand to do a proper job.

Chapter 5

The Destiny Project

Destiny is an automated program verifier for Java currently under development by the government. We have provided support to the Destiny project on two levels. We describe this support in more detail below, but we present a brief overview here.

General Support:

- We have read what documentation currently exists describing Destiny's logical foundations and operations. We have provided feedback on the contents of these documents and made suggestions for future work.
- We have used Destiny, and provided feedback and suggestions for the improvement of the user interface.

Specific Support:

- We have written a first version of a tool which reads Destiny's XML rule base or output, and converts the rules into ACL2.
- We have begun developing a library which will allow us to verify much of Destiny's rulebase.

5.1 Destiny Background

Destiny is an automated program verifier for Java currently under development by the government. Many of Destiny's operations are controlled by rules stored in an XML rule base. In addition, Destiny may produce XML output which requires verification by external theorem provers such as ACL2.

The correctness of Destiny's operations is dependent upon the correctness of the rules it uses. This rule base may be modified by the user. It is, therefore desirable to verify the correctness of these rules.

Destiny's rules come in several types. Each rule has an "action" associated with it. Some of these actions correspond to C++ code and are, therefore, not

amenable to direct verification. These rules tend to encode the state transitions of the JVM. They are therefore relatively stable and well-understood. Rules of this type are also very unlikely to be modified or added by a user of Destiny.

This is in contrast to rules whose action is "SUBR", or a SUBstitution Rule. These rules are likely to be about less well-understood aspects of Destiny's operations. Also, a user of Destiny is likely to add rules of this type. Luckily, these rules correspond directly to ACL2's notion of a rewrite rule, and are, therefore, amenable to verification.

5.2 Reading Destiny's XML

We have developed a first version of a tool which reads Destiny's XML rule base or output, and converts the rules into ACL2. The conversion from XML to ACL2 is relatively straight-forward. The tool that we have developed is flexible enough to easily track changes in Destiny's XML format as it is developed. It uses Franz's pxml utilities to parse the XML into lisp style s-expressions. We then translate the result into a format that ACL2 can understand. See Appendix F for an example of Destiny's XML and our translation.

5.3 Verifying Destiny's Rulebase

We have also begun developing a library to embed Destiny's logic into ACL2. We have a preliminary version of a book defining the predicate transformer logic as described in the paper "Predicate Transformers" by Frank Rimlinger. When completed these libraries will allow a greatly increased confidence in the soundness and correctness of Destiny. Already, in our first, partial, pass through the rule base we discovered several false rules.

The rules that we can verify fall in to two broad categories

- Rules about Destiny's predicate-transformer logic.
- Rules about Java arithmetic.

Destiny's predicate-transformer logic is described in a paper by Frank Rimlinger, *Predicate Transformers*. This logic describes the behavior of four types of objects — states, state-transformers, predicates, and predicate-transformers — and their associated operations. Three rules typical of this type follow.

```
(defthm dualcompose-1
  (implies (and (state-transformer-p x)
                (predicatep b))
           (ekual (compose (dual x) (dualp b))
                  (dualp (compose (pred b) x))))
  :doc "Changes composition of dual and a dualp to the dualp of a
        predicate composed with a state transition. Notice the order
        of X and b reverses, and the fact that b is a predicated can
```

```
      no longer be inferred from context."
  :rule-classes nil)

(defthm distributeand-r
  (implies (and (predicate-transformer-p z)
                (predicate-transformer-p x)
                (predicate-transformer-p y))
           (ekual (compose z (aand x y))
                  (aand (compose z x) (compose z y))))
  :doc "distributes and over compose in order to bring and to top
       level (left-hand rule for distributeAnd-2)"
  :rule-classes nil)

(defthm promoteandor-1
  (implies (or (and (predicatep a)
    (predicatep b)
    (predicatep c))
       (and (predicate-transformer-p a)
    (predicate-transformer-p b)
    (predicate-transformer-p c)))
   (ekual (oor (aand a b) c) (aand (oor a c) (oor b c))))
  :rule-classes nil)
```

And here is a rule about Java arithmetic.

```
(defthm promote-neg-times-1
  (implies (and (integerp x)
                (integerp y))
           (ekual (* (- x) y)
                  (- (* x y))))
  :rule-classes nil)
```

Appendix A

Three Sample Lemmatta

We present here three sample lemmatta we can now prove more easily. See also Appendix B for a larger and more complex example.

A.1 Example One

We present three example lemmatta which we can easily prove with the experimental version of ACL2 and its associated books. The first is a challenge problem that Matt Kauffman, one of the authors of ACL2, sent to the ACL2 mailing list a while ago. While he was able to prove it, he used eleven helper lemmatta to do so and was hoping someone had a library that would have made the job easier.

During our investigations we realized that his eleven lemmatta could be replaced with just one linear lemma. Unfortunately, this lemma is very sensitive to seemingly irrelevant details such as the addition of extra hypotheses and the naming of the variables. Our irritation with this state of affairs led us to seek a better solution and resulted in the first version of our non-linear extensions to the linear arithmetic package. Since then, this has been one of the standard test lemmatta we use whenever we modify the non-linear extensions.

```
(defthm arith-lemma-3  ;; As given by Matt Kauffman.
  (implies (and (<= a d)
                (<= (+ (- (* h a))
                       (* f a))
                    (+ (- (* h c))
                       (* g c)
                       b))
                (< i h)
                (< h g)
                (< e a)
                (force (<= g f))
                (force (rationalp a))
```

```
          (force (rationalp b))
          (force (rationalp c))
          (force (rationalp d))
          (force (rationalp e))
          (force (rationalp f))
          (force (rationalp g))
          (force (rationalp h))
          (force (rationalp i)))
     (<= (+ (- (* i e))
            (* e f))
         (+ (- (* i d))
            (* h d)
            (- (* h c))
            (* g c)
            b))))
```

A.2 Example Two

Our next example is a pair of proofs of one of the lemmatta from the quotient-remainder-lemmas book in the IHS library included with ACL2. The combination of more powerful rules for cancellation and the non-linear extensions allow us to prove the lemma directly. We present our version first:

```
(defthm floor-floor-integer
  (implies (and (rationalp x)
                (integerp i)
                (integerp j)
                (< 0 i)
                (< 0 j))
           (equal (floor (floor x i) j)
                  (floor x (* i j))))
;; The hints are needed for efficiency purposes only.
  :hints (("Goal" :in-theory (disable mod-linear-theory floor-linear-theory))))
```

We now present the original version from the IHS books for comparison:

```
(encapsulate ()

  ;;  This proof of FLOOR-FLOOR-INTEGER is an elaborate rewriting trick,
  ;;  which is spoiled by these 2 lemmas!

  (local (in-theory (disable rewrite-floor-mod rewrite-mod-mod)))

  ;;< These first 2 lemmas have nothing to do with the :LINEAR theory of
  ;;FLOOR and MOD, so we DISABLE the key :LINEAR lemmas to avoid thrashing.
```

```
(local (in-theory (disable floor-type-1 floor-type-2 floor-type-3
                           floor-type-4 floor-bounds mod-type mod-bounds)))

;; First, write x as a quotient and remainder of i*j.

(local
 (defthm floor-floor-integer-crock0
   (implies
    (and (rationalp x)
         (integerp i)
         (not (equal i 0))
         (integerp j)
         (not (equal j 0))
         (syntaxp (and (eq x 'x) (eq i 'i) (eq j 'j))))
    (equal (floor (floor x i) j)
           (floor (floor (+ (mod x (* i j))
                            (* (* i j) (floor x (* i j)))) i)
                  j)))
   :hints (("Goal" :in-theory (disable commutativity-2-of-+
                                       commutativity-2-of-*
                                       associativity-of-*)))))

;;  Next, divide out i and j through the sums.

(local
 (defthm floor-floor-integer-crock1
   (implies
    (and (rationalp x)
         (integerp i)
         (not (equal i 0))
         (integerp j)
         (not (equal j 0))
         (syntaxp (and (eq x 'x) (eq i 'i) (eq j 'j))))
    (equal (floor (floor x i) j)
           (+ (floor x (* i j)) (floor (floor (mod x (* i j)) i) j))))
   :hints
   (("Goal"
     :in-theory (disable floor-mod-elim)))))

(local
 (defthm floor-floor-integer-crock2
   (implies
    (and (rationalp x)
         (integerp i)
         (< 0 i)
         (integerp j)
```

```
          (< 0 j))
     (equal (floor (floor (mod x (* i j)) i) j)
            0))
    :hints (("Goal" :in-theory
             (set-difference-theories (enable floor-type-1
                                              floor-type-2
                                              floor-type-3
                                              mod-type)
                                      '(floor-bounds mod-bounds
                                                     <-*-left-cancel
                                                     <-*-/-left-commuted))
      :use ((:instance floor-bounds (x (mod x (* i j))) (y i))
            (:instance mod-bounds (x x) (y (* i j)))
            (:instance <-*-left-cancel
                       (z (/ i)) (x (mod x (* i j))) (y (* i j))))))))

;; Voila!

(defthm floor-floor-integer
  (implies
   (and (integerp i)
        (integerp j)
        (< 0 i)
        (< 0 j)
        (rationalp x))
   (equal (floor (floor x i) j)
          (floor x (* i j)))))
```

A.3 Example Three

Our final example is one illustrating the MMIX interpreter. This interpreter was written before we decided to standardize on the JVM in order to take advantage of other work going on in the ACL2 community.

We prove the correctness of a simple shift and add multiplier. The experimental version of ACL2 and its associated books will prove this without any hints or helper lemmatta.

```
(defun shift-and-add ()
;; \$3 <-- \$2 * \$1
  '(     (NEGU \$3 0 0)       ;; \$3 <-- 0
    LOOP (BZ \$2 HALT)        ;; Branch if zero
         (BEV \$2 EVEN)       ;; Branch if even
         (ADDU \$3 \$3 \$1)   ;; \$3 <-- \$3 + \$1
         (SRU \$2 \$2 1)      ;; Shift \$2 right 1
         (SLU \$1 \$1 1)      ;; Shift \$1 left 1
```

```
        (JMP LOOP)          ;; Jump
    EVEN (ADDU \$3 \$3 0)    ;; No-op
        (SRU \$2 \$2 1)     ;; Shift \$2 right 1
        (SLU \$1 \$1 1)     ;; Shift \$1 left 1
        (JMP LOOP)          ;; Jump
        (HALT)))

(defthm shift-and-add-is-multiply
  (implies (and (integerp x) (<= 0 x)
                (integerp y) (<= 0 y) (< y (expt2 64))
                (integerp a) (<= 0 a)
                (equal meas (lngth y))
                (< (+ a (* x y)) (expt2 64))
                (< (* x (expt2 (length y))) (expt2 64)))
           (equal (run (make-state 0
                                   '((\$1 . ,x)
                                     (\$2 . ,y)
                                     (\$3 . ,a))
                                   nil
                                   nil
                                   (shift-and-add)
                                   )
                       (cplus 1 (cplus (ctimes meas 6) 2)))
                  (make-state 11
                              '((\$1 . ,(* x (expt2 (lngth y))))
                                (\$2 . 0)
                                (\$3 . ,(* x y)))
                              nil
                              nil
                              (shift-and-add)
                              ))))
```

Appendix B

Legato's Multiplier

I need to make sure that this still works as given. I should also see if I can do better.

B.1 Introduction

This ACL2 script solves a challenge posed by Bill Legato, to prove that a program written for the Mostek 6502 microprocessor correctly implements multiplication. The program is described in "A Weakest Precondition Model for Assembly Language Programs" by Bill Legato, dated June 19, 2000.

B.2 Proof Script

```
(include-book "/var/local/Arithmetic-4/Bind-free/top")
(include-book "/var/local/Arithmetic-4/Floor-mod/floor-mod")


(set-non-linearp t)


;;;;;;;;;;;;;;;;;;;;;;;;


;; I now define the mult-loop function.  The use of temp is not
;; necessary, but I find it a little easier to read this way.
;; Getting rid of temp does not affect the proof.

;; I also prefer the use of
;; (if (equal (mod f1 2) 1)
;;     (+ a f2)
;;   a)
```

```
;; instead of the equivalent
;; (+ (* (mod f1 2) (+ a f2))
;;    (* (- 1 (mod f1 2)) a)).
;; Using the second of these, an extra lemma is needed.

;; However, I give two definitions, each of which will work with the
;; script.

(defun dec (x i)
  (if (zp x)
      (+ -1 (expt 2 i))
    (+ -1 x)))

(defun mult-loop (f1 f2 a low c x i)
  ;; This function is a generalization of the multiplier to work
  ;; with any width register.
  ;; The variable i is the width of the registers.
  (cond ((or (not (integerp x))
             (<= x 0)
             (not (integerp i))
             (<= i 0))
         (mv f1 f2 a low c x))
        ((equal x 1)
         (let ((temp (if (equal (mod f1 2) 1)
                         (+ a f2)
                       a)))
           (mv (+ (* c (expt 2 (+ -1 i))) (floor f1 2))
               f2
               (floor temp 2)
               (+ (* (expt 2 (+ -1 i)) (mod temp 2)) (floor low 2))
               (mod low 2)
               (- x 1))))
        (t
         (let ((temp (if (equal (mod f1 2) 1)
                         (+ a f2)
                       a)))
           (mult-loop (+ (* c (expt 2 (+ -1 i))) (floor f1 2))
                      f2
                      (floor temp 2)
                      (+ (* (expt 2 (+ -1 i)) (mod temp 2)) (floor low 2))
                      (mod low 2)
                      (dec x i)
                      i)))))
#|
(defun mult-loop (f1 f2 a low c x i)
  (cond ((or (not (integerp x))
```

```
            (<= x 0))
         (mv f1 f2 a low c x))
        ((equal x 1)
         (mv (+ (* c (expt 2 (+ -1 i))) (floor f1 2))
             f2
             (floor (+ (* (mod f1 2) (+ a f2))
                       (* (- 1 (mod f1 2)) a))
                    2)
             (+ (* (expt 2 (+ -1 i))
                   (mod (+ (* (mod f1 2) (+ a f2))
                           (* (- 1 (mod f1 2)) a))
                        2))
                (floor low 2))
             (mod low 2)
             (- x 1)))
        (t
         (mult-loop (+ (* c (expt 2 (+ -1 i))) (floor f1 2))
                    f2
                    (floor (+ (* (mod f1 2) (+ a f2))
                              (* (- 1 (mod f1 2)) a))
                           2)
                    (+ (* (expt 2 (+ -1 i))
                          (mod (+ (* (mod f1 2) (+ a f2))
                                  (* (- 1 (mod f1 2)) a))
                               2))
                       (floor low 2))
                    (mod low 2)
                    (- x 1)
                    i))))

;; Here is the extra lemma which is needed with this version of
;; mult-loop.

(defthm crock-0
    (implies (and (integerp i)
                  (integerp a)
                  (< a (expt 2 i))
                  (integerp f2)
                  (< f2 (expt 2 i))
                  (integerp x))
             (< (floor (+ a (* f2 (mod x 2))) 2)
                (expt 2 i))))
|#

;; This definition is not really needed, but it makes things a little
;; easier to read.
```

```
(defun mult-prog (m n low c i)
  (mult-loop m n 0 low c 8 i))


;;;;;;;;;;;;;;;;;;;;;;;;;

;; I always like to test my functions.


(defun test-prog (m n)
  (mv-let (f1 f2 a low c x)
    (mult-prog m n 0 0 8)
    (declare (ignore f1 f2 c x))
    (equal (+ (* 256 a) low)
           (* m n))))

(test-prog 4 5)

(test-prog 157 83)

(test-prog 126 214)


;;;;;;;;;;;;;;;;;;;;;;;;;


;; These next two definitions are also not really needed, but they
;; make things a little easier to read.

(defun byte-p (x i)
  (and (integerp x)
       (<= 0 x)
       (< x (expt 2 i))))

(defun bit-p (x)
  (or (equal x 0)
      (equal x 1)))


;; We now prove the four helper lemmatta needed for this proof.


;; I used to think that this first one was a little too expensive
;; for general use, but I have now changed my mind.  I have had to
;; include it too many times anyway.  It will be in future versions
```

```
;; of my library.

(defthm crock-1
    (implies (and (integerp a)
                  (integerp b))
             (integerp (* a b))))


;; These next three lemmatta are needed becuase I had to turn off
;; the non-linear arithmetic for the proof of loop-thm below.  If you
;; run the script without them, you will see that they are all fairly
;; obvious lemmatta to add.  This also applies to crock-0 above.  If I had
;; more patience (or a much faster computer) I might not need any
;; of these.

(defthm crock-2
    (implies (and (integerp i)
                  (integerp x)
                  (< x (expt 2 i)))
             (< (+ (floor x 2)
                   (expt 2 (+ -1 i)))
                (expt 2 i))))

(defthm crock-3
    (implies (and (integerp i)
                  (integerp x)
                  (integerp a)
                  (< x (expt 2 i)))
             (< (+ (floor x 2)
                   (* (expt 2 (+ -1 i)) (mod a 2)))
                (expt 2 i))))

(encapsulate
 ()

 ;; The use of crock-4-crock (or rather seeing that it could be used)
 ;; depends on knowing more about the workings of linear arithmetic
 ;; than I would expect most users to have.  I therefore give a
 ;; more obvious proof below.

 (local
  (defthm crock-4-crock
      (implies (and (integerp x) (< 0 x))
               (integerp (* 1/2 (expt 2 x))))
    :rule-classes :type-prescription))
```

```
 (defthm crock-4
     (implies (and (integerp x)
                   (case-split (< 0 x))
                   (integerp k)
                   (< k (expt 2 (+ -1 x))))
              (< (+ 1 (* 2 k)) (expt 2 x))))

 )
#|
(encapsulate
 ()

 (local
  (defun ind-hint (x)
    (if (zp x)
        t
      (ind-hint (+ -1 x)))))

 (local
  (defthm crock-4-crock
      (implies (and (integerp x)
                    (<= 0 x)
                    (integerp k)
                    (< k (expt 2 x)))
               (< (+ 1/2 k) (expt 2 x)))
    :hints (("Goal" :induct (ind-hint x)))
    :rule-classes nil))

 (defthm crock-4
     (implies (and (integerp x)
                   (case-split (< 0 x))
                   (integerp k)
                   (< k (expt 2 (+ -1 x))))
              (< (+ 1 (* 2 k)) (expt 2 x)))
   :hints (("Goal" :use (:instance crock-9-crock
                                   (x (+ -1 x))))))
 |#

 (set-bind-free-error nil)

 (encapsulate
 ()

 ;; The use of crock-4-crock (or rather seeing that it could be used)
 ;; depends on knowing more about the workings of linear arithmetic
 ;; than I would expect most users to have.  I therefore give a
```

```
;; more obvious proof below.

(local
 (defthm crock-4-crock
     (implies (and (integerp x) (< 0 x))
              (integerp (* 1/2 (expt 2 x))))
   :rule-classes :type-prescription))

(defthm crock-4
    (implies (and (integerp x)
                  (case-split (< 0 x))
                  (< k (expt 2 (+ -1 x)))
                  (integerp k))
             (< (+ 1 (* 2 k)) (expt 2 x)))
:rule-classes ((:rewrite) (:linear :trigger-terms ((expt 2 x)))))

)


;; I had to turn off non-linear arithmetic for the proof of loop-thm.
;; It was just too expensive, and I did not have the patience to
;; let it run.  The proof takes quite a while as it is.

;(set-non-linearp nil)

(defthm loop-thm
    (implies (and (byte-p f1 i)
                  (byte-p f2 i)
                  (byte-p a i)
                  (byte-p low i)
                  (bit-p c)
                  (byte-p x i)
                  (< 0 x)
                  (<= x i)
                  (integerp i)
                  (< 0 i))
             (equal (mv-let (new-f1 new-f2 new-a new-low new-c new-x)
                      (mult-loop f1 f2 a low c x i)
                      (declare (ignore new-f1 new-f2 new-c new-x))
                      (+ (* (expt 2 i) new-a) new-low))
                    (+ (floor (+ (* (expt 2 i) a) low) (expt 2 x))
                       (* f2 (mod f1 (expt 2 x)) (expt 2 (- i x))))))
  :hints (("Goal" :do-not '(generalize)
                  :induct t))
  :rule-classes nil)
```

```
(set-non-linearp t)

(defthm main
    (implies (and (byte-p f1 8)
                  (byte-p f2 8)
                  (byte-p low 8)
                  (bit-p c))
             (equal (mv-let (new-f1 new-f2 new-a new-low new-c new-x)
                      (mult-prog f1 f2 low c 8)
                      (declare (ignore new-f1 new-f2 new-c new-x))
                      (+ (* 256 new-a) new-low))
                    (* f1 f2)))
  :hints (("Goal" :in-theory (disable mult-loop)
                  :use ((:instance loop-thm
                                   (a 0)
                                   (x 8)
                                   (i 8))))))
```

Appendix C

Extended Metafunctions Examples

We have used this new type of metafunction to perform simplifications which would not have been possible before. Some examples follow. (Note that much of what is in this appendix has been superseded in actual use by bind-free rules.)

1. We have written a rule which will cancel 'like' factors from both sides of an inequality only when doing so will not cause a case-split; i.e., one which when given

```
(thm (implies (and (rationalp x) (< x 1)
                   (rationalp y) (< 0 y)
                   (acl2-numberp z) (< 0 z))
              (< (* a x y z)
                 (* b x y z))))
```

   will only cancel `y` from both sides. Although `x`, `y`, and `z` all occur on both sides of the inequality, calling `mfc-ts` on each of these will reveal that only the `y` is a candidate for cancelation. Cancelling `x` would induce a case-split based on its sign, while `z` is not known to be rational and so cannot be cancelled[1].

2. It is a theorem that

```
(implies (and (rationalp x)
              (rationalp y)
              (rationalp n)
              (integerp (/ x n))
```

[1] Complex numbers are, unfortunately, provided with a linear order in ACL2. This causes difficulties since `0 < i` and `(* i 2) < (* i -2 i)`, but `2 < (* -2 i)` is false. At least one of the authors feels that this is one of the few true mistakes made in the design of ACL2.

```
      (not (equal n o)))
 (equal (mod (+ x y) n)
        (mod y n))).
```

Previously, an ever-increasing number of variations of the above theorem would be required to catch all instances of interest[2] (and some of potential interest may never be seen since they may have arisen as hypotheses which were never relieved).

We have written a sequence of rules which put increasing amounts of effort into finding cancellations such as the above. The first scans the type alist looking for candidates for cancellation. This will find most of the ones which are 'staring it in the face'. The next one also calls type-set via `mfc-ts` on the quotient of each addend with the modulus. This is not too expensive and should pick up a few more cancellations than the first, but it will still miss some due to the weakness of type reasoning. The final meta rule calls `mfc-rw` on each quotient and is the most powerful of the three, but may be too expensive for ordinary use. It was the desire to write this sequence of rules, and the impossibility of expressing them, which led to the creation of extended meta rules.

[2] What about `(mod (+ x y z) n)` when `(integerp (/ (+ x z) n))` is true by hypothesis?

Appendix D

Meta-Integerp

D.1 Introduction

We include the source for an example extended meta rule, `meta-integerp-correct`. Before writing this meta rule and its associated metafunction, the use of a helper lemma such as one of the following three had been required at various points during our work.

```
(defthm  intp-1-2-1
 (implies (and (integerp (+ x z))
               (integerp y))
          (integerp (+ x y z))))  ;; (1)

(defthm  nintp-1-2-1-2
 (implies (and (integerp (+ w y))
               (rationalp (+ x z))
               (not (integerp (+ x z))))
          (not (integerp (+ w x y z)))))  ;; (2)

(defthm  intp-2-3-1-3-2
 (implies (and (integerp n)
               (integerp (+ a y))
               (integerp (+ b x)))
          (integerp (+ a b n x y))))  ;; (3)
```

The metafunction `meta-integerp` attempts to rewrite terms of the form `(integerp (+ x y z))`, where z may itself be a sum, and is designed to operate in conjunction with a couple of simple rewrite rules. We illustrate this by showing how the metafunction would rewrite each of the above marked terms.

```
(1) (integerp (intp-+ (hide (binary-+ x z))
                      (hide (fix y))))
```

```
(2) (integerp (intp-+ (hide (binary-+ x z))
                      (hide (binary-+ w y))))

(3) (integerp (intp-+ (hide (binary-+ a y))
                      (hide (intp-+ (hide (binary-+ b x))
                                    (hide (fix n))))))
```

The function `hide` is the identity and is used here for efficiency purposes, while `intp-+` is a 'dummy' function which means the same thing as + and is used both to prevent ACL2 from undoing things and to format the term correctly for the rewrite rules which follow.

```
(defthm intp-1
  (implies (and (integerp x)
                (integerp y))
           (integerp (intp-+ (hide x) (hide y)))))

(defthm n-intp-1
  (implies (and (acl2-numberp x)
                (not (integerp x))
                (integerp y))
           (not (integerp (intp-+ (hide x) (hide y))))))

(defthm n-intp-2
  (implies (and (integerp x)
                (acl2-numberp y)
                (not (integerp y)))
           (not (integerp (intp-+ (hide x) (hide y))))))
```

Since these theorems will be all that ACL2 knows about `intp-+`, the desired result (here, the correctness of the three above marked conclusions) will be obtained. We have found that this division of labor between metafunction and rewrite rule makes writing and proving the correctness of metafunctions easier.

D.2 The Code Listing

First, we need to define a couple of helper functions:

```
(defabbrev fn-symb (x)
  (and (consp x)
       (car x)))

(defmacro arg1 (x)
  `(cadr ,x))

(defmacro arg2 (x)
  `(caddr ,x))
```

The next three functions will tear apart or build up the sum under consideration:

```
(defun leaves (term)

;; We gather all the addends of a sum into a list.

  (cond ((endp term)
         ''0)
        ((eq (fn-symb term)
             'BINARY-+)
         (if (eq (fn-symb (fargn term 2))
                 'BINARY-+)
             (cons (fargn term 1)
                   (leaves (fargn term 2)))
           (list (fargn term 1)
                 (fargn term 2))))
        (t
         (list term))))

(defun tree (leaves)

;; Leaves is a list of addends.  We return a term representing
;; their sum.

  (cond ((endp leaves)
         ''0)
        ((endp (cdr leaves))
         `(FIX ,(car leaves)))
        ((null (cddr leaves))
         (list 'BINARY-+ (car leaves) (cadr leaves)))
        (t
         (list 'BINARY-+ (car leaves) (tree (cdr leaves))))))

(defun big-tree (bags)

;; Bags is a list of lists of addends.  We return a term
;; representing the sum, but formatted such that the theorems
;; intp-1, n-intp-1, and n-intp-2 can do their work.

  (cond ((endp bags)
         ''0)
        ((endp (cdr bags))
         (tree (car bags)))
        ((endp (cddr bags))
         `(INTP-+ (HIDE ,(tree (car bags)))
                  (HIDE ,(tree (cadr bags)))))
```

```
      (t
       '(INTP-+ (HIDE ,(tree (car bags)))
                (HIDE ,(big-tree (cdr bags)))))))
```

The following function gathers type information on each leaf (addend) by calling `type-set`.

```
(defun bag-leaves-ts (leaves intp-bags non-intp-bags
                             context state)

;; This function is called only by bag-leaves.  Upon initial
;; entry, intp-bags and non-intp-bags are nil, leaves are the
;; addends of a sum, and context and state are as supplied by
;; the metafunction meta-integerp.  We call type-set on each
;; of the leaves, and for each one:
;; 1. If it is known to be an integer, we form a bag consisting
;;    of that leaf and accumulate it into intp-bags.
;; 2. Similarly, if it is known to be a non-integer acl2-number,
;;    we accumulate it into non-intp-bags.

  (if (endp leaves)
      (mv intp-bags non-intp-bags)
    (let ((leaf-type (mfc-ts (car leaves) context state)))
      (cond ((ts-subsetp leaf-type *ts-integer*)
             (bag-leaves-ts (cdr leaves)
                            (cons (list (car leaves)) intp-bags)
                            non-intp-bags
                            context state))
            ((and (ts-subsetp leaf-type *ts-acl2-number*)
                  (ts-subsetp leaf-type
                              (ts-complement *ts-integer*)))
             (bag-leaves-ts (cdr leaves)
                            intp-bags
                            (cons (list (car leaves)) non-intp-bags)
                            context state))
            (t
             (bag-leaves-ts (cdr leaves)
                            intp-bags non-intp-bags
                            context state))))))
```

This function walks through the `type-alist` looking for useful information.

```
(defun bag-leaves-type-alist (type-alist intp-bags non-intp-bags)

;; This function is called only by bag-leaves.  Upon initial
;; entry, intp-bags and non-intp-bags contain the information
;; found by bag-leaves-ts.  We now scan down the type-alist
```

```
;; and form bags for any terms known to be integers or not,
;; accumulating them into the appropriate list.

  (cond ((endp type-alist)
         (mv intp-bags non-intp-bags))
        ((not (equal (fn-symb (caar type-alist))
                     'BINARY-+))
         (bag-leaves-type-alist (cdr type-alist)
                                intp-bags non-intp-bags))
        ((ts-subsetp (cadr (car type-alist))
                     *ts-integer*)
         (bag-leaves-type-alist (cdr type-alist)
                                (cons (leaves (caar type-alist))
                                      intp-bags)
                                non-intp-bags))
        ((ts-subsetp (cadr (car type-alist))
                     (ts-complement *ts-integer*))
         (bag-leaves-type-alist (cdr type-alist)
                                intp-bags
                                (cons (leaves (caar type-alist))
                                      non-intp-bags)))
        (t
         (bag-leaves-type-alist (cdr type-alist)
                                intp-bags non-intp-bags))))

(defun bag-leaves (leaves context state)
  (mv-let (intp-bags non-intp-bags)
    (bag-leaves-ts leaves nil nil context state)
    (bag-leaves-type-alist (mfc-type-alist context)
                           intp-bags non-intp-bags)))

(defun subtract-leaf (leaf leaves)

;; We try to remove leaf from the list leaves.  We return two
;; values, a flag indicating whether we were successful and,
;; if so, the remaining leaves.

  (cond ((endp leaves)
         (mv nil nil))
        ((equal leaf (car leaves))
         (mv t (cdr leaves)))
        (t
         (mv-let (flag new-leaves)
           (subtract-leaf leaf (cdr leaves))
           (if flag
               (mv t (cons (car leaves)
```

```
                        new-leaves))
            (mv nil nil))))))

(defun subtract-bag (bag leaves)

;; We try to remove a bag of leaves from the list leaves.

  (cond ((endp bag)
         (mv t leaves))
        ((endp (cdr bag))
         (subtract-leaf (car bag) leaves))
        (t
         (mv-let (flag new-leaves)
           (subtract-leaf (car bag) leaves)
           (if flag
               (subtract-bag (cdr bag) new-leaves)
             (mv nil nil))))))
```

The next two functions make use of the information gathered above.

```
(defun collect-bags-intp (leaves intp-bags)

;; This function is called only by collect-bags.  Leaves is a list of
;; the addends of the sum, and intp-bags is a list of bags.  Each bag
;; contains a collection of terms whose sum is known to be an integer.
;; We try to find a cover for leaves consisting of the bags in
;; intp-bags.  If we are succesful, the sum is an integer.
;; We return two values, a flag indicating whether we were
;; successful, and if so a list of the bags which form the cover.

  (cond ((endp leaves)
         (mv t nil))
        ((endp intp-bags)
         (mv nil nil))
        (t
         (mv-let (flag new-leaves)
           (subtract-bag (car intp-bags) leaves)
           (if flag
               (mv-let (flag new-bags)
                 (collect-bags-intp new-leaves (cdr intp-bags))
                 (if flag
                     (mv t (cons (car intp-bags)
                                 new-bags))
                   (collect-bags-intp leaves (cdr intp-bags))))
             (collect-bags-intp leaves (cdr intp-bags)))))))

(defun collect-bags-non-intp (leaves intp-bags non-intp-bags)
```

```
;; This function is called only by collect-bags.  We try to
;; find a cover for leaves consisting of one of the non-intp-bags
;; and the bags in intp-bags.
;; We return two values, a flag indicating whether we were
;; successful, and if so a list of the bags which form the cover.

  (cond ((endp non-intp-bags)
         (mv nil nil))
        (t
         (mv-let (flag new-leaves)
           (subtract-bag (car non-intp-bags) leaves)
           (if flag
               (mv-let (flag bag-list)
                 (collect-bags-intp new-leaves intp-bags)
                 (if flag
                     (mv t
                         (cons (car non-intp-bags)
                               bag-list))
                   (collect-bags-non-intp leaves
                                          intp-bags
                                          (cdr non-intp-bags))))
             (collect-bags-non-intp leaves
                                    intp-bags
                                    (cdr non-intp-bags)))))))

(defun collect-bags (leaves intp-bags non-intp-bags)
  (mv-let (flag bag-list)
    (collect-bags-non-intp leaves intp-bags non-intp-bags)
    (if flag
        (mv flag bag-list)  ;; non-intp
      (collect-bags-intp leaves intp-bags))))
```

We now define the metafunction itself.

```
(defun meta-integerp (term context state)

;; Assumptions: 1. Term is right-associated.  2. Not all leaves
;; are known to be integers by type-set.

  (if (and (eq (fn-symb term)
               'INTEGERP)
           (eq (fn-symb (arg1 term))
               'BINARY-+)
           (eq (fn-symb (arg2 (arg1 term)))
               'BINARY-+))
```

```
;; We have a term of the form:
;; (integerp (+ x y z)).

      (let ((leaves (leaves (fargn term 1))))
        (mv-let (intp-bags non-intp-bags)
          (bag-leaves leaves context state)
          (mv-let (flag bags)
            (collect-bags leaves intp-bags non-intp-bags)
            (if flag
                `(INTEGERP ,(big-tree bags))
              term))))
    term))

(defthm meta-integerp-correct
  (equal (intp-eva term a)
         (intp-eva (meta-integerp term context state) a))
  :rule-classes ((:meta :trigger-fns (INTEGERP))))
```

D.3 Conclusion

Note that the functions above can be divided into three groups

1. those which tear down and build up terms,
2. those which gather information,
3. those which use this information,

and that all they really accomplish is the rearrangement of a term into a form which some simple, custom, rewrite rules can then exploit. We have found this to be a very helpful division which makes the writing, verification, and modification of meta rules easier.

Appendix E

Bind-Free Examples

We give examples of the use of bind-free hypotheses from the perspective of a user interested in reasoning about arithmetic, but it should be clear that bind-free can be used for many other purposes also.

E.1 Example One — Cancel a Common Factor

```
(defun bind-divisor (a b)

; If a and b are polynomials with a common factor c, we return a
; binding for x.  We could imagine writing get-factor to compute the
; gcd, or simply to return a single non-invertible factor.


  (let ((c (get-factor a b)))
    (and c (list (cons 'x c)))))


(defthm cancel-factor
  ;; We use case-split here to ensure that, once we have selected
  ;; a binding for x, the rest of the hypotheses will be relieved.
  (implies (and (acl2-numberp a)
                (acl2-numberp b)
                (bind-free (bind-divisor a b) (x))
                (case-split (not (equal x 0)))
                (case-split (acl2-numberp x)))
           (iff (equal a b)
                (equal (/ a x) (/ b x)))))
```

E.2 Example Two — Pull Integer Summand out of Floor

```
(defun fl (x)
  ;; This function is defined, and used, in the IHS books.
  (floor x 1))


(defun int-binding (term mfc state)
  ;; The call to mfc-ts returns the encoded type of term.
  ;; Thus, we are asking if term is known by type reasoning to
  ;; be an integer.
  (declare (xargs :stobjs (state) :mode :program))
  (if (ts-subsetp (mfc-ts term mfc state)
                  *ts-integer*)
      (list (cons 'int term))
  nil))


(defun find-int-in-sum (sum mfc state)
  (declare (xargs :stobjs (state) :mode :program))
  (if (and (nvariablep sum)
           (not (fquotep sum))
           (eq (ffn-symb sum) 'binary-+))
      (or (int-binding (fargn sum 1) mfc state)
          (find-int-in-sum (fargn sum 2) mfc state))
    (int-binding sum mfc state)))
```

; Some additional work is required to prove the following. So for ; purposes of illustration, we wrap skip-proofs around the defthm.

```
(skip-proofs
 (defthm cancel-fl-int
  ;; The use of case-split is probably not needed, since we should
  ;; know that int is an integer by the way we selected it.  But this
  ;; is safer.
   (implies (and (acl2-numberp sum)
                 (bind-free (find-int-in-sum sum mfc state) (int))
                 (case-split (integerp int)))
            (equal (fl sum)
                   (+ int (fl (- sum int)))))
   :rule-classes ((:rewrite :match-free :all)))
)
```

; Arithmetic libraries will have this sort of lemma.

```
(defthm hack (equal (+ (- x) x y) (fix y)))
```

```
(in-theory (disable fl))


(thm (implies (and (integerp x) (acl2-numberp y))
              (equal (fl (+ x y)) (+ x (fl y)))))
```

E.3 Example Three — Simplify Terms Such as (equal (+ a (* a b)) 0)

```
(defun factors (product)
  ;; We return a list of all the factors of product.  We do not
  ;; require that product actually be a product.
  (if (eq (fn-symb product) 'BINARY-*)
      (cons (fargn product 1)
            (factors (fargn product 2)))
    (list product)))


(defun make-product (factors)
  ;; Factors is assumed to be a list of ACL2 terms.  We return an
  ;; ACL2 term which is the product of all the ellements of the
  ;; list factors.
  (cond ((atom factors)
         ''1)
        ((null (cdr factors))
         (car factors))
        ((null (cddr factors))
         (list 'BINARY-* (car factors) (cadr factors)))
        (t
         (list 'BINARY-* (car factors) (make-product (cdr factors))))))


(defun quotient (common-factors sum)
  ;; Common-factors is a list of ACL2 terms.   Sum is an ACL2 term each
  ;; of whose addends have common-factors as factors.  We return
  ;; (/ sum (make-product common-factors)).
  (if (eq (fn-symb sum) 'BINARY-+)
      (let ((first (make-product (set-difference-equal (factors (fargn sum 1))
                                                       common-factors))))
        (list 'BINARY-+ first (quotient common-factors (fargn sum 2))))
    (make-product (set-difference-equal (factors sum)
                                        common-factors))))
```

```
(defun intersection-equal (x y)
  (cond ((endp x)
         nil)
        ((member-equal (car x) y)
         (cons (car x) (intersection-equal (cdr x) y)))
        (t
         (intersection-equal (cdr x) y))))


(defun common-factors (factors sum)
  ;; Factors is a list of the factors common to all of the addends
  ;; examined so far.  On entry, factors is a list of the factors in
  ;; the first addend of the original sum, and sum is the rest of the
  ;; addends.  We sweep through sum, trying to find a set of factors
  ;; common to all the addends of sum.
  (declare (xargs :measure (acl2-count sum)))
  (cond ((null factors)
         nil)
        ((eq (fn-symb sum) 'BINARY-+)
         (common-factors (intersection-equal factors (factors (fargn sum 1)))
                         (fargn sum 2)))
        (t
         (intersection-equal factors (factors sum)))))


(defun simplify-terms-such-as-a+ab-rel-0-fn (sum)
  ;; If we can find a set of factors common to all the addends of sum,
  ;; we return an alist binding common to the product of these common
  ;; factors and binding quotient to (/ sum common).
  (if (eq (fn-symb sum) 'BINARY-+)
      (let ((common-factors (common-factors (factors (fargn sum 1))
                                            (fargn sum 2))))
        (if common-factors
            (let ((common (make-product common-factors))
                  (quotient (quotient common-factors sum)))
              (list (cons 'common common)
                    (cons 'quotient quotient)))
          nil))
    nil))


(defthm simplify-terms-such-as-a+ab-=-0
  (implies (and (bind-free
                 (simplify-terms-such-as-a+ab-rel-0-fn sum)
```

```
                (common quotient))
               (case-split (acl2-numberp common))
               (case-split (acl2-numberp quotient))
               (case-split (equal sum
                                  (* common quotient))))
          (equal (equal sum 0)
                 (or (equal common 0)
                     (equal quotient 0)))))


(thm (equal (equal (+ u (* u v)) 0)
     (or (equal u 0) (equal v -1))))
```

Appendix F

Destiny XML to ACL2 Example

This could still be cleaned up, but it is not bad (I hope).

In this appendix we present a snippet of Destiny's XML rulebase as it existed on May 13th 2003, the output of our tool on this snippet, and the final rewrite rules as verified in ACL2.

F.1 XML Rulebase Snippet

Here is a snippet of Destiny's XML rulebase. This snippet encodes the two substitution rules `dualCompose-1` and `distributeAnd-R`, two rules typical of those that we have verified. The lines beginning with ";;" are not part of the original file, but are comments we have added for purposes of description.

```
<rule>

;; The name of the rule:

        <name>dualCompose-1</name>

;; It's associated action, in this case a SUBstitution Rule.

        <action>SUBR</action>

;; A brief English description of the rules purpose or use.  Note that
;; this was one long line in the original.

        <description>Changes composition of dual and a dualp to the
                     dualp of a predicate composed with a state
                     transition. Notice the order of X and b reverses,
```

```
                    and the fact that b is a predicated can no longer
                    be inferred from context.
        </description>

;; Destiny's rules, just like ACL2's, can be active/enabled or not.

        <active>1</active>

;; The variables used in the rule.  Here the variables are b and X.
;; Both are unconstrained, meaning that they can pattern-match with
;; anything.  Posible constraints are opStack and number, among many
;; others, which constrain the pattern-match to specific types of
;; Destiny objects.  At present, we ignore this aspect of Destiny's
;; rulebase, but
;; 1. Its use is largely resticted to rules with actions other than
;; substitution.
;; 2. The proper place to deal with such issues is in the hypotheses
;; which is the next XML piece.

        <variable_array>
                <variable>
                        <name>b</name>
                        <requirement>unconstrained</requirement>
                </variable>
                <variable>
                        <name>X</name>
                        <requirement>unconstrained</requirement>
                </variable>
        </variable_array>

;; Destiny does not yet use hypotheses.  This is a weak part in the
;; Destiny to ACL2 link.  See our comments about the final ACL2
;; rules below.

        <hypothesis><root/></hypothesis>

;; Destiny uses a system of encoding the pattern and substitution
;; of a rule which we do not attempt to explain.  Suffice it to say
;; that it is simpler to decode than it appears at first sight.

        <pattern>
                <root>790</root>
                <k>
                        <id>790</id>
                        <r>18</r>
                        <al>
```

```
                        <h>791</h>
                        <h>792</h>
                </al>
        </k>
        <k>
                <id>791</id>
                <r>32</r>
                <al>
                        <h>2</h>
                </al>
        </k>
        <k>
                <id>792</id>
                <r>793</r>
                <al>
                        <h>7</h>
                </al>
        </k>
        <p>
                <id>793</id>
                <in/>
                <st>aliased_rule_keyType</st>
                <n>dualp</n>
                <bs>794</bs>
        </p>
        <s>
                <id>794</id>
                <in/>
                <st>rule_key</st>
                <n>dualp</n>
        </s>
</pattern>
<substitution>
<root>795</root>
        <k>
                <id>795</id>
                <r>793</r>
                <al>
                        <h>796</h>
                </al>
        </k>
        <k>
                <id>796</id>
                <r>18</r>
                <al>
                        <h>797</h>
```

```
                                <h>2</h>
                        </al>
                </k>
                <k>
                        <id>797</id>
                        <r>798</r>
                        <al>
                                <h>7</h>
                        </al>
                </k>
                <p>
                        <id>798</id>
                        <in/>
                        <st>aliased_rule_keyType</st>
                        <n>pred</n>
                        <bs>799</bs>
                </p>
                <s>
                        <id>799</id>
                        <in/>
                        <st>rule_key</st>
                        <n>pred</n>
                </s>
        </substitution>
</rule>

;; The next rule begins here.

<rule>
        <name>distributeAnd-R</name>
        <action>SUBR</action>
        <description>distributes and over compose in order to bring
                     and to top level (left-hand rule for
                     distributeAnd-2)
        </description>
        <active>1</active>
        <variable_array>
                <variable>
                        <name>x</name>
                        <requirement>unconstrained</requirement>
                </variable>
                <variable>
                        <name>y</name>
                        <requirement>unconstrained</requirement>
                </variable>
                <variable>
```

```
                <name>z</name>
                <requirement>unconstrained</requirement>
        </variable>
</variable_array>
<hypothesis><root/></hypothesis>
<pattern>
        <root>800</root>
        <k>
                <id>800</id>
                <r>18</r>
                <al>
                        <h>628</h>
                        <h>801</h>
                </al>
        </k>
        <k>
                <id>801</id>
                <r>802</r>
                <al>
                        <h>33</h>
                        <h>574</h>
                </al>
        </k>
        <p>
                <id>802</id>
                <in/>
                <st>aliased_rule_keyType</st>
                <n>and</n>
                <bs>803</bs>
        </p>
        <s>
                <id>803</id>
                <in/>
                <st>rule_key</st>
                <n>and</n>
        </s>
</pattern>
<substitution>
<root>804</root>
        <k>
                <id>804</id>
                <r>802</r>
                <al>
                        <h>805</h>
                        <h>806</h>
                </al>
```

```
            </k>
            <k>
                    <id>805</id>
                    <r>18</r>
                    <al>
                            <h>628</h>
                            <h>33</h>
                    </al>
            </k>
            <k>
                    <id>806</id>
                    <r>18</r>
                    <al>
                            <h>628</h>
                            <h>574</h>
                    </al>
            </k>
    </substitution>
```

F.2 Tool Output

Here is the raw output of our tool for translating Destiny's XML into ACL2. Note that, one again, we broke the comment up from one long line.

```
(defthm DUALCOMPOSE-1
  (implies nil
           (ekual (COMPOSE (DUAL X) (DUALP B))
                  (DUALP (COMPOSE (PRED B) X))))
  :doc "Changes composition of dual and a dualp to the dualp of a
        predicate composed with a state transition. Notice the order
        of X and b reverses, and the fact that b is a predicated can
        no longer be inferred from context."
  :rule-class nil)

(defthm DISTRIBUTEAND-R
  (implies nil
           (ekual (COMPOSE Z (AAND X Y))
                  (AAND (COMPOSE Z X) (COMPOSE Z Y))))
  :doc "distributes and over compose in order to bring and to top
        level (left-hand rule for distributeAnd-2)"
  :rule-class nil)
```

F.3 The Final ACL2 Rules

Here are the two rules in their final form as verified by ACL2. Note that we had to determine the correct hypotheses by hand. Destiny will shortly support the use of hypotheses.

```
(defthm dualcompose-1
  (implies (and (state-transformer-p x)
                (predicatep b))
           (ekual (compose (dual x) (dualp b))
                  (dualp (compose (pred b) x))))
  :doc "Changes composition of dual and a dualp to the dualp of a
        predicate composed with a state transition. Notice the order
        of X and b reverses, and the fact that b is a predicated can
        no longer be inferred from context."
  :rule-classes nil)

(defthm distributeand-r
  (implies (and (predicate-transformer-p z)
                (predicate-transformer-p x)
                (predicate-transformer-p y))
           (ekual (compose z (aand x y))
                  (aand (compose z x) (compose z y))))
  :doc "distributes and over compose in order to bring and to top
        level (left-hand rule for distributeAnd-2)"
  :rule-classes nil)
```

Appendix G

Assume-true-false

We here present a mini-essay on our work to improve the internal ACL2 function `assume-true-false`. This essay was originally written to document the new code, and so is directed towards someone who is reading it in company with the code. It appears in the file `type-set-b.lisp` and can be found by searching for the string "mini-essay".

G.1 Mini-essay on Assume-true-false-if

Normally, when ACL2 rewrites one of the branches of an `IF` expression, it "knows" the truth or falsity (as appropriate) of the test. More precisely, if `x` is a term of the form `(IF test true-branch false-branch)`, when ACL2 rewrites `true-branch`, it can determine that `test` is true by type reasoning alone, and when it rewrites `false-branch` it can determine that `test` is false by type reasoning alone. This is certainly what one would expect.

However, previously, if test was such that it would print as `(AND test1 test2)` – so that `x` would print as `(IF (AND test1 test2) true-branch false-branch)` – when ACL2 rewrote `true-branch` it only knew that `(AND test1 test2)` was true — it did not know that `test1` was true nor that `test2` was true, but only that their conjunction was true. There were also other situations in which ACL2's reasoning was weak about the test of an IF expression.

The function `assume-true-false-if` was written to correct this problem but it caused other problems of its own. This mini-essay records one of the difficulties encountered and its solution.

In initial tests with the new `assume-true-false-if`, more than three-fourths of the books distributed with ACL2 failed to certify. Upon examination it turned out that ACL2 was throwing away many of the :use hints as well as some of the results from generalization rules. Let us look at a particular example (from inequalities.lisp in the arithmetic library):

```
(defthm <-*-right-cancel
  (implies (and (rationalp x)
```

```
                (rationalp y)
                (rationalp z))
           (iff (< (* x z) (* y z))
                (cond
                 ((< 0 z)
                  (< x y))
                 ((equal z 0)
                  nil)
                 (t (< y x)))))
  :hints (("Goal" :use
           ((:instance (:theorem
                        (implies (and (rationalp a)
                                      (< 0 a)
                                      (rationalp b)
                                      (< 0 b))
                                 (< 0 (* a b))))
                       (a (abs (- y x)))
                       (b (abs z)))))))
```

This yields the subgoal:

```
(IMPLIES (IMPLIES (AND (RATIONALP (ABS (+ Y (- X))))
                       (< 0 (ABS (+ Y (- X))))
                       (RATIONALP (ABS Z))
                       (< 0 (ABS Z)))
                  (< 0 (* (ABS (+ Y (- X))) (ABS Z))))
         (IMPLIES (AND (RATIONALP X)
                       (RATIONALP Y)
                       (RATIONALP Z))
                  (IFF (< (* X Z) (* Y Z))
                       (COND ((< 0 Z) (< X Y))
                             ((EQUAL Z 0) NIL)
                             (T (< Y X)))))))
```

Previously, the preprocess-clause ledge of the waterfall would see this as

```
((NOT (IMPLIES (IF (RATIONALP (ABS (BINARY-+ Y (UNARY-- X))))
                   (IF (< '0 (ABS (BINARY-+ Y (UNARY-- X))))
                       (IF (RATIONALP (ABS Z))
                           (< '0 (ABS Z))
                           'NIL)
                       'NIL)
                   'NIL)
               (< '0
                  (BINARY-* (ABS (BINARY-+ Y (UNARY-- X)))
                            (ABS Z)))))
 (IMPLIES (IF (RATIONALP X)
```

```
            (IF (RATIONALP Y) (RATIONALP Z) 'NIL)
            'NIL)
          (IFF (< (BINARY-* X Z) (BINARY-* Y Z))
               (IF (< '0 Z)
                   (< X Y)
                   (IF (EQUAL Z '0) 'NIL (< Y X))))))
```

and return

```
(((NOT (IF (IF (RATIONALP (ABS (BINARY-+ Y (UNARY-- X))))
               (IF (< '0 (ABS (BINARY-+ Y (UNARY-- X))))
                   (IF (RATIONALP (ABS Z))
                       (< '0 (ABS Z))
                       'NIL)
                   'NIL)
               'NIL)
           (IF (< '0
                  (BINARY-* (ABS (BINARY-+ Y (UNARY-- X)))
                            (ABS Z)))
               'T
               'NIL)
           'T))
  (NOT (RATIONALP X))
  (NOT (RATIONALP Y))
  (NOT (RATIONALP Z))
  (IF (< (BINARY-* X Z) (BINARY-* Y Z))
      (IF (IF (< '0 Z)
              (< X Y)
              (IF (EQUAL Z '0) 'NIL (< Y X)))
          'T
          'NIL)
      (IF (IF (< '0 Z)
              (< X Y)
              (IF (EQUAL Z '0) 'NIL (< Y X)))
          'NIL
          'T))))
```

Now, under the old regime, when rewrite got hold of the conclusion of the :use hint,

```
(< '0
(BINARY-* (ABS (BINARY-+ Y (UNARY-- X)))
          (ABS Z)))
```

it would would know that X, Y, and Z were rational and that the IF expression

```
(IF (RATIONALP (ABS (BINARY-+ Y (UNARY-- X))))
 (IF (< '0 (ABS (BINARY-+ Y (UNARY-- X))))
```

```
    (IF (RATIONALP (ABS Z))
        (< '0 (ABS Z))
        'NIL)
    'NIL)
'NIL)
```

was true. But it would not know, for instance, that (< '0 (ABS (BINARY-+ Y (UNARY-- X)))) was true.

With the introduction of `assume-true-false-if`, however, ACL2 would know that each element of the conjunction represented by the IF expression was true, and so would be able to determine that the conclusion of the `:use` hint was true by type-reasoning alone (since the original `:theorem` was so proved). Thus, the whole hint rewrites to true and is removed. Bummer.

Previously, the conclusion of a `:use` hint or of a fact derived from a generalization rule would, in affect, be rewritten (the first time) under the type-alist of the overall goal to be proved. We now handle IMPLIES differently in order to return to this original behavior.

The basic idea is not to expand IMPLIES except within rewrite where we can easily maintain the proper type-alists. Two simple exceptions to this rule are in tautologyp and distribute-first-if. We expand IMPLIES within tautologyp for obvious reasons and within distribute- first-if because earlier tests (presumably still valid) showed that this was both faster and more fruitful. Note that this second exception implies that an IMPLIES will never show up due to a user defined function.

A slightly more subtle exception is in preprocess-clause where we expand an IMPLIES if it is derived from the original theorem to be proved. This is necessary to provide a context for rewrite (See the comments in `preprocess-clause beginning` "Note: Once upon a time (in Version 1.5)" for more on this.

Appendix H

README

This is the README from arithmetic-4.

H.1 How to Prove Theorems Using ACL2

H.1.1 Introduction

When using ACL2, it is important to take an organized approach to the proof process. Sections 1.B. and 1.C. are a summary of the material to be found in Chapter 9 of Computer-Aided Reasoning: An Approach and Chapter 6 of Computer-Aided Reasoning: ACL2 Case Studies. I highly recommend these two books to the serious ACL2 user. In section 1.D. we address a few issues that are too often overlooked by the ACL2 user.

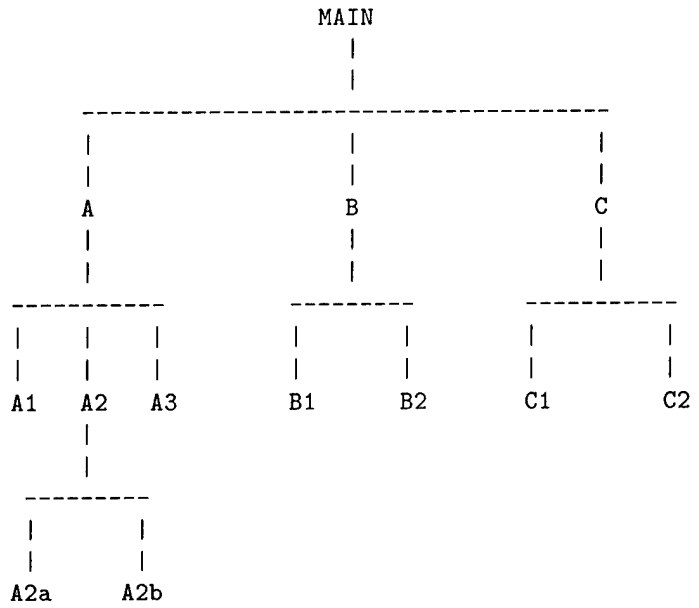
Here, we mention a few simple points about using ACL2.

1. ACL2 is automatic only in the sense that you cannot steer it once it begins a proof attempt. You can only interrupt it.
2. You are responsible for guiding it, usually by proving the necessary lemmas.
3. Never prove a lemma without also thinking about how it will be used.
4. To lead ACL2 to a proof, you usually must know the proof yourself.

H.1.2 How to Use the Theorem Prover — "The Method"

In this section, we outline 'The Method". While it is only one of many possible styles of working with ACL2, we have found it to be a useful starting point from which to develop one's own method. In the following section we will discuss some of its weaknesses and how to get around them.

Let us imagine that we want to prove some theorem, MAIN; that in order to do so it is sufficient to use lemmas A, B, and C; and that each of these lemmas can be proved using sub-lemmas as depicted in the proof tree below.



One will not usually have worked out the proof (or even necessity) of every lemma before trying to prove MAIN. In fact, most users work out the detailed structure of the proof tree during their interaction with ACL2. Merely keeping track of what has been proved and what remains to be proved is daunting enough. It is this book-keeping task that "The Method" is designed to assist.

The goal of the procedure we outline here is to produce a sequence of defthm commands which will lead ACL2 to a proof of MAIN. The procedure will produce a post-order traversal of the tree (A1, A2a, A2b, A2, A3, A, B1, B2, B, C1, C2, C, MAIN).

We use ACL2 in conjunction with a text editor such as Emacs where we can run ACL2 as a process in a *shell* buffer, and maintain a second buffer for input commands, referred to as the script buffer.

When we are done, the script buffer will contain the postorder traversal of the proof tree. But, as we construct the proof the script buffer is logically divided into two parts by an imaginary line we call the barrier. The part above the barrier consists of the commands that have been carried out, while the part below the buffer contains the commands we intend to carry out later. The barrier is, by convention, denoted by the s-expression (I-am-here).

Initially, the script buffer should contain only the theorem MAIN with the barrier at the top of the buffer; i.e., the done list is empty and the to-do list contains only MAIN. Here is "The Method".

1. Think about the proof of the first theorem in the to-do list. Have the necessary lemmas been proved? If so, go to step 2. Otherwise add them to the front of the to-do list and repeat step 1.

2. Try proving the first theorem with ACL2 and let the output stream into the *shell* buffer. Abort the proof if it runs too long.

3. If ACL2 succeeded, advance the barrier past the successful command and go to step 1. Otherwise go to step 4.

4. Inspect the output of the failed proof attempt from the beginning rather than from the end, You should look for the first place the proof attempt deviated from your imagined proof. Modify the script appropriately — this usually means adding lemmas to the front of the to-do list although you may need to add hints to the current theorem. Sometimes, especially if the reason for the failure is that the theorem is false, this may mean modifying the script both above and below the barrier. Go to step 1.

H.1.3 Structured Theory Development.

There are several shortcomings of "The Method". In this section, we describe some of these shortcomings and discuss an elaboration of "The Method" which should lessen their influence.

First, "The Method" provides no guidance for developing the over-all structure of a substantial proof effort. It is too easy to lose one's way in the middle of the proof effort and, moreover, once the proof is complete it can be quite difficult to comprehend its structure. Such comprehension is important for presenting the proof to others, and is also useful for modifying the proof — either in order to clean it up or in order to prove a related theorem.

Second, use of "The Method" is prone to lead to the following: One desires to prove a certain lemma, but it requires a lemma in support of its proof, which leads to another lemma to be proved in support of *that* one, and so on. At some point the effort seems misguided, but by then the evolving proof structure is far from clear and it is difficult to decide how far to back up. Even if a decision is made to back up to a particular lemma, is it clear which lemmas already proved may be discarded?

Finally, even if the above process is successful for a while, it can ultimately be problematic in an even more frustrating way. Suppose that one attempts to prove some goal theorem, and from the failed proof attempt one identifies rewrite rules that appear to be helpful, say, L1 and L2. Suppose further that additional rewrite rules are proved on the way to proving L1 and L2. When one again attempts to prove the original goal theorem, those additional rules can send the proof attempt in a new direction and prevent L1 and L2 from being used.

We describe here a modular, top-down methodology which reflects common proof development practice in the mathematical community and is designed to lessen the above described problems.

Here is an outline describing many proofs, both mechanically checked ones, and others.

1. To prove the main theorem Main:

2. It should suffice to prove lemmas A, B, and C. (Main Reduction)

3. We need to draw upon a body of other, previously completed work. (Proof Support)

4. We may also need a few additional, minor, lemmas. (Proof Hacking)

The outline may be reflected in the following structure of a top-level book, which (at least initially) can use defaxiom or skip-proofs as shown in order to defer the proofs of the main lemmas.

```
(include-book ''lib'')        ; 3. Support

(defaxiom A ...)              ; 2. Main Reduction
(defaxiom B ...)
(defaxiom C ...)

<minor lemmas>                ; 4. Proof Hacking

(defthm MAIN ...)             ; 1. Goal
```

This use of defaxiom has the advantage that one can verify that lemmas A, B, and C are actually the ones needed to prove MAIN without having to do a lot of work to prove them. In step four of our description of "The Method", we said that one should add any new lemmas to the front of the to-do list and then go directly to step one. Here, we recommend adding these new lemmas using skip-proofs (or as axioms) and making sure that they are what is needed before continuing.

For purposes of illustration assume that we next want to prove lemma C. Above, we suggested that this be done by adding any necessary sub-lemmas in front of this main lemma. Here, we suggest that one of two techniques be used. If the proof of C requires only two or three lemmas, and each of these requires no further sub-lemmas, prove C within the scope of an encapsulate as in:

```
(encapsulate
  ()

  (local
    (defthm C1
      ...))

  (local
    (defthm C2
      ...))

  (defthm C
    ...)
```

```
).
```

Note that by making lemmas C1 and C2 local to the encapsulate they will not be seen outside of the context in which they were designed to be used, and so will not change the behavior of any events later in the script. Their influence has been limited and the chances for surprises has been lessened.

If the proof of lemma C is more complex — if it requires more than a couple of lemmas or if those sub-lemmas themselves require sub-lemmas — lemma C should be proved in a separate book named C.lisp and the following should appear in the main script:

```
(encapsulate
  ()

  (local (include-book "C"))

  (defthm C
    ...)

  ).
```

The book C.lisp should then be recursively treated as described here with lemma C playing the role of MAIN. This delegation of the work to prove C to another book makes the file MAIN.lisp much easier to read, and the hierarchy of book inclusions reflects the overall structure of the proof. (See certify-book and include-book in the documentation. We assume here that C.lisp will be certified at some point during the proof development cycle.)

Note that we are using two types of books:

1. Lemma Books: Book name is the same as the name of the final theorem proved in the book. Main lemmas can be postponed to be proved in subsidiary lemma books, temporarily replaced by defaxiom in the present book.

2. Library Books: Other than a lemma book, typically it contains generally useful definitions and lemmas.

Our top-down methodology suggests a focus on developing reasonably short lemma books. The trick to keeping their length under control is first to identify the main lemmas in support of the book's goal theorem, then pushing their proofs into subsidiary lemma sub-books, especially when supporting sub-lemmas may be required. Each main lemma is handled as illustrated above with sub-books A, B, and C: an encapsulate event contains first a local include-book of the corresponding lemma sub-book, and second the main lemma.

An important aspect of this approach is that the way in which a sub-lemma is proved in such a sub-book will not affect the certification of the parent book.

That is, the use of local around the include-book of a lemma sub-book allows the sub-book to be changed, other than changing its goal theorem, without affecting the certification of the parent book. This modularity can prevent replay problems often encountered when using less structured approaches to mechanically-assisted proof.

Although our focus here has been on lemma books, there is still a role for library books. It can be useful from time to time to browse ones current collection of books and to pull out the most general of these lemmas and put them into one or more library books. This is how the library books in books/arithmetic, for instance, were developed.

H.1.4 A Few Notes.

We finish this part with some more simple points which are easy to overlook:

1. Definitions, both recursive and non-recursive, should be disabled as soon as possible. Non-recursive functions, which can be regarded as abbreviations, will be opened up by ACL2 at the first opportunity and so will rarely be used as intended if they are enabled.

   Recursive definitions will not disappear in the same way, and so the desirability of disabling them is not as readily apparent. However, if they are enabled ACL2 will open them up each time they are seen, attempt to rewrite them, and then close them back up again unless the rewritten definition is "better" than the unopened definition. Unless one is proving simple facts about the function this is often a large waste of time. We have seen proofs go from taking more than 30 minutes to under 15 seconds by disabling (an admittedly large number of complicated and mutually) recursive function definitions.

2. In addition to the "obvious" facts about a function which one should prove before disabling it, one should prove lemmas about the simple cases which can arise surprisingly often. In the case of arithmetic functions this would include such cases as repeated arguments, e.g., (logand x x) = x, and base cases, e.g., (expt x 0) = (if (equal x 0) 0 1), (expt x 1) = (fix x), and (expt x -1) = (/ x).

3. Once one is comfortable with the use of rewrite rules, one should next explore the use of type-prescription rules and linear lemmas. Type-prescription rules can be a bit tricky to get used to, but it is well worth the effort.

   One common problem newer users encounter is with the way hypotheses of type-prescription rules are relieved — type reasoning alone must be sufficient. The primitive types in ACL2 are:

```
*TS-ZERO*                  ;;; {0}
*TS-POSITIVE-INTEGER*      ;;; positive integers
*TS-POSITIVE-RATIO*        ;;; positive non-integer rationals
```

```
*TS-NEGATIVE-INTEGER*       ;;; negative integers
*TS-NEGATIVE-RATIO*         ;;; negative non-integer rationals
*TS-COMPLEX-RATIONAL*       ;;; complex rationals
*TS-NIL*                    ;;; {nil}
*TS-T*                      ;;; {t}
*TS-NON-T-NON-NIL-SYMBOL*   ;;; symbols other than nil, t
*TS-PROPER-CONS*            ;;; null-terminated non-empty lists
*TS-IMPROPER-CONS*          ;;; conses that are not proper
*TS-STRING*                 ;;; strings
*TS-CHARACTER*              ;;; characters.
```

If ones hypotheses deal only with these types (or there union such as integerp or true-listp) one is usually OK. Otherwise, the necessary facts must be present explicitly in the hypotheses of the goal being proved, or else themselves be relievable by other type-prescription rules. In particular, rewriting is not used. (For example, the presence of (¡ 4 x) as a hypothesis is not sufficient for type-reasoning to establish (¡ 3 x).)

4. The use of defaxiom (or skip-proofs) is a good way to avoid wasting time proving useless lemmas.

5. False theorems are surprisingly hard to prove. Even the most experienced ACL2 user will regularly write a false theorem. When a proof just will not go through, examine the output of the failed proof attempt with an eye to test cases which may reveal a problem.

6. (From the documentation)

   Stack overflows are most often caused by looping rewrite rules. In some Lisps, especially GCL, stack overflows often manifest themselves as segmentation faults, causing the entire ACL2 image to crash. Finding looping rewrite rules can be tricky, especially if you are using books supplied by other people.

   A wonderful trick is the following. When there is a stack overflow during a proof, abort and then try it again after turning on rewrite stack monitoring with :brr t. When the stack overflows again, exit to raw Lisp. How you exit to raw Lisp depends on which Lisp you are using. In Allegro Common Lisp, for example, the stack overflow will leave you in an interactive break. You must exit this break, e.g., with :continue 1. This will leave you in the top-level ACL2 command loop. You must exit this environment with :q. That will leave you in raw Allegro.

   After getting into raw Lisp, execute `(cw-gstack *deep-gstack* state)`

   The loop in the rewriter will probably be evident!

   If you are in GCL the stack overflow will probably cause a segmentation fault and abort the Lisp job. This makes it harder to debug but here is what you do. First, re-create the situation just prior to submitting

the form that will cause the stack overflow. You can do this without suffering through all the proofs by using the :ld-skip-proofsp option of ld to reload your scripts. Before you submit the form that causes the stack overflow, exit the ACL2 command loop with :q. In raw GCL type `(si::use-fast-links nil)`

This will slow GCL down but make it detect and signal stack overflows rather than overwrite the system memory. Now reenter the ACL2 command loop with `(lp)`.

Now carry on as described above, turning on rewrite stack monitoring with :brr t and provoking the stack overflow. When it occurs, you will be in an interactive break. Exit to raw Lisp with two successive :q's, one to get out of the error break and the next to get out of the top-level ACL2 command loop. Then in raw GCL execute the cw-gstack above.

Suggestion: Once you have found the loop and fixed it, you should execute the ACL2 command :brr nil, so that you don't slow down subsequent proof attempts. If you are in GCL, you should also get into raw Lisp and execute `(si::use-fast-links t)`.

H.2 How to Use the Arithmetic-4 Books

In brief: First certify the books in arithmetic-3. Then, after starting ACL2, execute the following two forms:

```
(include-book "/<library-path>/arithmetic-3/Bind-free/top")
(set-non-linearp t)
```

If you are reasoning about floor and/or mod also execute:

```
(include-book "/<library-path>/arithmetic-3/Floor-mod/floor-mod")
```

Note: Occasionally non-linear arithmetic will go off into the weeds and consume too much time. If you notice that ACL2 is taking too long on a particular goal or theorem, you can try executing (set-non-linearp nil) before retrying the problematic theorem. This will result in a theory which "knows" less about multiplication, so one should examine any failed goals with this in mind.

H.2.1 Choosing a Normal Form.

The major goal of this library is to provide a set of rewrite rules which will drive systems of arithmetic expressions into a useful normal form. There are, however, at least two barriers to achieving this goal. First, as Kurt Gödel showed, integer arithmetic is formally undecidable; i.e., there is no algorithm which will fully normalize all systems of arithmetic expressions. Second, there is no reason to believe that there is one "best" normal form. We therefore provide the ability to choose any one of several alternative normal forms.

Before we go any further we should mention that there is, in general, no reason to use the information in the rest of this section. We believe that the default setup is sufficient for most purposes. On occasion one will run across a lemma which requires a different normal form and it is for these occasions that we provide the following; however, for many users it will be sufficient to just include the appropriate books and to execute (set-non-linearp t) as described above.

Gather/Scatter-exponents: There are (at least) two ways of normalizing the (almost) equivalent terms:

```
(expt x (+ m n)) and
(* (expt x m) (expt x n)).
```

(Question: Under what conditions are these two terms not equivalent?)

One can choose the first of these as the normal form and gather exponents, or one can choose the second as the normal form and scatter exponents. That is, one can choose to rewrite `(expt x (+ m n))` to `(* (expt x m) (expt x n))` — to scatter exponents; or to rewrite `(* (expt x m) (expt x n))` to `(expt x (+ m n))` — to gather exponents. By default, exponents are gathered when the book Bind-free/top is loaded.

To switch to scattering exponents, execute `(scatter-exponents)` at the top level prompt. To resume gathering exponents, execute `(gather-exponents)`. These two forms are macros which expand to the appropriate set of commands. (To see their expansions, one can execute

```
:trans (scatter-exponents) or
:trans (gather-exponents) at the top-level.
```

It is slightly more complex to switch theories within a :hints form. One can use

```
:in-theory (e/d (scatter-exponents-theory)
                (gather-exponents-theory))
```

and

```
:in-theory (e/d (gather-exponents-theory)
                (scatter-exponents-theory
                 prefer-positive-exponents-scatter-exponents-theory))
```

to switch to scattering exponents or gathering exponents respectively.

Prefer-positive-addends: This theory (enabled by default) moves negative addends to the other side of an equality or inequality. A simple example is:

```
(< (+ a (- b) c)
   (+ d (* -3 e)))
 ===>
(< (+ a c (* 3 e))
   (+ b d)).
```

Execute (do-not-prefer-positive-addends) at the top-level to disable this theory and (prefer-positive-addends) to re-enable it. To switch theories within a :hints form, one can use:

```
:in-theory (disable prefer-positive-addends-theory)
```

and

```
:in-theory (enable prefer-positive-addends-theory)
```

to disable or enable this theory.

Prefer-positive-exponents: This theory (disabled by default) moves factors with negative exponents to the other side of an equality or inequality. A simple example is:

```
(equal (* a (/ b) c)
       (* d (expt e -3)))
===>
(equal (* a c (expt e 3))
       (* b d)).
```

One can enable this with (prefer-positive-exponents) and disable it with (do-not-prefer-positive-exponents) at the top level prompt.

Note that (prefer-positive-exponents) is a specialization of (scatter-exponents). It therefore switches to scattering exponents if this has not been done previously. Thus, one may need to execute (gather-exponents) after (do-not-prefer-positive-exponents) if one was originally gathering exponents and wished to resume doing so. To switch theories within a :hints form, one can use:

```
:in-theory (e/d (scatter-exponents-theory
                 prefer-positive-exponents-scatter-exponents-theory)
                (gather-exponents-theory))
```

and

```
:in-theory (disable prefer-positive-exponents-scatter-exponents-theory)
```

or

```
:in-theory (e/d (gather-exponents-theory)
                (scatter-exponents-theory
                 prefer-positive-exponents-scatter-exponents-theory)).
```

H.2.2 Mini-theories.

The book mini-theories contains several useful lemmatta not enabled by default. We recommend that you look briefly at this book. Here we describe two of the lemmas and a macro which are included.

Two lemmas — `|(- x)|` and `|(/ x)|` — are included for those who like to replace subtraction with multiplication by -1 and division by exponentiation. If you enable these, be sure to disable their inverses, `|(* -1 x)|` and `|(expt x -1)|`, or you will get infinite rewrite loops.

We also include the macro e/d in mini-theories. Sometimes one wishes to both enable certain rules and disable others within a single :in-theory hint. This macro makes it easier. Examples:

```
(e/d (lemma1 lemma2))                ;Equivalent to (ENABLE lemma1 lemma2).
(e/d () (lemma))                     ;Equivalent to (DISABLE lemma).
(e/d (lemma1) (lemma2 lemma3))       ;ENABLE lemma1 then DISABLE lemma2 and
                                     ;lemma3.
(e/d () (theory1) (theory2))         ;DISABLE theory1 then ENABLE theory2.

(e/d (|(- x)| |(/ x)|)               ;ENABLE |(- x)| and |(/ x)| then
     (|(* -1 x)| |(expt x -1)|))     ;DISABLE |(* -1 x)| and |(expt x -1)|
```

Appendix I

Non-Linear Arithmetic

I.1 abstract

The ACL2 theorem prover has been extended to automatically verify sets of polynomial inequalities that include nonlinear relationships. In this paper we describe our mechanization of linear and nonlinear arithmetic in ACL2. The nonlinear arithmetic procedure operates in cooperation with the pre-existing ACL2 linear arithmetic decision procedure, which is based on Presburger arithmetic. It extends what can be automatically verified with ACL2, thereby eliminating the need for certain types of rules in ACL2's database while simultaneously increasing the performance of the ACL2 system when verifying arithmetic conjectures. The resulting system lessens the human effort required to construct a large arithmetic proof by reducing the number of intermediate lemmas that must be proven to verify a desired theorem.

I.2 Introduction

Mechanical theorem proving or proof checking systems offer a rigorous methodology with which to structure and check proofs. Each such system offers a different degree of automation — directly affecting its capability and ease of use. We have extended the ACL2 theorem proving system [7, 8, 9] with an automated verification procedure that enhances the linear arithmetic decision procedure. ACL2 can now more easily verify sets of inequalities containing nonlinear arithmetic relationships.

In this paper we describe our mechanization of linear and nonlinear arithmetic in ACL2. Before doing so however, we briefly describe the theory behind the procedures and provide a couple of examples of their use. Inequalities can be combined by cross-multiplication and addition to permit the deduction of an additional inequality. For example, if $0 < poly1$ and $0 < poly2$, and c and d are positive rational constants, then $0 < c \cdot poly1 + d \cdot poly2$. Here, we are using the facts that multiplication by a positive rational constant does not change the sign

of a polynomial and that the sum of two positive polynomials is itself positive. This is linear arithmetic. Also, given the above, we have $0 < c \cdot poly1 \cdot poly2$. In this nonlinear case, we are using the fact that the product of two positive polynomials is itself positive.

Now suppose we want to prove

$$3 \cdot x + 7 \cdot a < 4 \quad \wedge \quad 3 < 2 \cdot x \quad \Longrightarrow \quad a < 0.$$

We can do this by assuming the two hypotheses and the negation of the conclusion, and looking for a contradiction. We therefore start with the three inequalities:

$$0 < -3 \cdot x + -7 \cdot a + 4 \tag{I.1}$$

$$0 < 2 \cdot x + -3 \tag{I.2}$$

$$0 \leq a. \tag{I.3}$$

We cross-multiply and add the first two — that is, multiply inequality (I.1) by two and inequality (I.2) by three, and then add the respective sides. This yields

$$0 < -14 \cdot a + -1. \tag{I.4}$$

Note that the new inequality does not mention x. If we choose two inequalities with the same leading term and leading coefficients of opposite sign, we can generate an inequality in which that leading term is not present. This is the general strategy employed by the linear arithmetic decision procedure.

If we next cross-multiply and add inequality (I.4) with inequality (I.3), we get

$$0 < -1, \tag{I.5}$$

a false polynomial. We have, therefore, proved our theorem.

This process illustrated above of cross-multiplying and adding two inequalities will be referred to as "cancelling" the two inequalities. We shall refer to such obviously false inequalities as (I.5) as "contradictions," and speak of any process that results in one of these as "generating a contradiction."

Next, suppose that we have the three assumptions

$$3 \cdot x \cdot y + 7 \cdot a < 4 \quad \text{or} \quad 0 < -3 \cdot x \cdot y + -7 \cdot a + 4 \tag{I.6}$$

$$3 < 2 \cdot x \quad \text{or} \quad 0 < 2 \cdot x + -3 \tag{I.7}$$

$$1 < y \quad \text{or} \quad 0 < y + -1, \tag{I.8}$$

and we wish to prove that $a < 0$. We proceed by assuming the negation of our goal, $0 <= a$, and looking for a contradiction.

Note that in this case no two inequalities above have a leading term in common. In this situation there are no cancellations which can be performed. However, (I.6) has a product as its leading term, $x \cdot y$, and for each of the factors of that product, x and y, there is an inequality which has such a factor as a

leading term. When nonlinear arithmetic is enabled, ACL2 will multiply[1] (I.7) and (I.8), obtaining

$$0 < 2 \cdot x \cdot y + -3 \cdot y + -2 \cdot x + 3. \tag{I.9}$$

The addition of this polynomial will allow cancellation to continue[2] and, in this case, we will prove our goal. Thus, just as ACL2 adds two polynomials together when they have the same largest unknown of opposite signs in order to create a new smaller polynomial, ACL2 can now multiply polynomials together when the product of their largest unknowns is itself the largest unknown of another polynomial.

I.2.1 Related Work and Plan of the Paper

It is often desirable to verify the correct operation of computer hardware or software. These operations may involve arithmetic, as in the floating-point hardware of a modern microprocessor or pointer arithmetic in a C program.

Several approaches to automating the verification of arithmetic lemmas have been tried. Great progress has been made as is illustrated by the many substantial proofs recently completed in PVS, HOL, and ACL2 [10, 5, 13]. The existing state-of-the-art is, however, not sufficient. The level of user expertise and effort required for the above-mentioned work is too high.

One of the primary difficulties encountered has been the fact that the formulae to be proved are rarely limited to just the four basic arithmetic operations, $+$, $-$, $*$, and $/$, but often involve diverse semantic constructs or, at the least, user-defined functions. Theorem provers, therefore, cannot limit themselves to "pure" arithmetic but must work with combinations of theories.

Our approach has developed from an engineering, results-oriented perspective, and we have therefore concentrated on decreasing the user's effort for the types of lemmas we see ACL2 users attempting to prove. Others have taken a more theoretical approach, whereby they can guarantee algorithmic completeness[3] over an exactly specified domain.

Several groups have built such systems by combining small-domain specific provers. Nelson and Oppen [11] and Shostak [14] describe frameworks with which one can combine separate, existing, decision procedures into one larger procedure. This work has been extended by others; e.g., by Kapur [6] and SRI [12]. While this approach has some nice properties such as completeness and efficiency, it can be somewhat limiting. Some of these limitations arise because the procedures to be integrated are treated as fixed black-boxes. Armando and Ranise [1], describe a method for augmenting the black-boxes. Other limitations

[1] How to multiply two inequalities will be described below.

[2] Inequality I.9 can be canceled with I.6. The result can be canceled with I.8, and so on. The final cancellation will be with the negation of our goal.

[3] A decision procedure is said to be complete if it always returns a (correct) answer when asked to verify a true theorem. An incomplete procedure, by contrast, may return an "I don't know" or even not return at all.

arise from concerns over efficiency. Harrison [4] explores the use of a full decision procedure over the reals and discusses its desirability.

We build on earlier work by Boyer and Moore [2] and share a common design philosophy with theirs. We regard ACL2's various procedures as a mutually recursive nest of functions, and have tuned both the interfaces and internals of these procedures using feedback from users to guide the process. It is also similar to work by Cyrluk and Kapur [3]; they too were concerned with augmenting existing linear arithmetic decision procedures to handle nonlinear inequalities, and their design was also driven by engineering concerns. We have the advantage of possessing much faster computers than were available at that time and believe that the time has come to reexamine the feasibility of more ambitious, but still incomplete algorithms, for handling nonlinear arithmetic. In this paper we present our first attempt at fully integrating a nonlinear arithmetic semi-decision procedure into ACL2. We present merely an outline of our work, and do not discuss nor even mention many of the heuristics that we have employed to limit and guide our algorithms.

In the next section, we provide the background required for the rest of the paper. This includes the definitions of polys, pots, and pot-lists, as well as a short discussion of type reasoning and linearization. In the following sections we describe the subprocedures that make up the linear and nonlinear arithmetic procedures. The linear arithmetic procedure consists of two nested loops. The innermost of these, the linear arithmetic decision procedure, is described in Section I.4. This procedure is responsible for adding inequalities to the pot-list. In Section I.5 we present linear arithmetic's outer loop, the linear lemmas procedure. This procedure attempts to gather additional inequalities, in order to allow further cancellations.

The nonlinear arithmetic procedure consists of three nested loops. The innermost of these is the same as for linear arithmetic. The next one, described in Section I.6, is an augmentation of that described in Section I.5. Nonlinear arithmetic's outer loop is presented in Section I.7. We conclude with a few remarks about the labor that can now be saved.

I.3 Background

The procedures we will describe here operate on polynomial inequalities. A "polynomial" is a sum of terms, each of which is either a rational constant or the product of a rational constant and an "unknown." An example polynomial is $3 \cdot x + -7/2 \cdot a + 2$; here x and a are the unknowns. The unknowns, however, need not be variable symbols — e.g., $|x|$, x^n, or $f(x, y)$ may be used as unknowns. Thus, $-3 \cdot |x| + a$ is also a polynomial.

A "polynomial inequality", or a "poly" for short, is an inequality (either $<$ or $\leq$) between 0 and a polynomial; e.g., $0 < 3 \cdot x + -7/2 \cdot a + 2$ and $0 \leq -3 \cdot |x| + a$ are polys. We refer to obviously false polys such as $0 < 0$ as "contradictions".

Polys are stored in groups called "pots". All the polys with the same largest

unknown[4] are stored in a single pot, which is said to be "labeled by" or "about" that unknown. These pots are further divided into two compartments — one for "positive" polys (those with a positive leading coefficient) and one for "negative" polys (those with a negative leading coefficient). A pot represents the conjunction of the polys in it.

The pots are stored in a "pot-list". The pot-list represents the conjunction of the pots in it. An example[5] pot-list is:

label	positives	negatives
b	$0 \leq b + -1 \cdot a + -1$	
$f(x, y)$	$0 \leq f(x, y)$	$0 < -1 \cdot f(x, y) + -1 \cdot b + a$

We refer to the b and $f(x, y)$ above as their pots' "labels."

The procedures that we will describe here all take, among other arguments, a pot-list and a list of polys to be added to the pot-list. They return either an augmented pot-list or a contradiction — the latter case indicating success as in Section I.2.

I.3.1 Type Reasoning and Polys from Type-set

We shall treat type reasoning — carried out by calling the ACL2 function `type-set` — as something of a black-box. For present purposes only a few things need be known about it.

First, we use it here to quickly answer the question "To what arithmetic category does this expression belong?" where the possible answers include (but are not limited to) zero,[6] positive integer, non-negative integer, or rational.

Second, we can sometimes form polys about an expression based on the answer given. For example, if x is said to be a negative rational, we can create the poly $0 < -1 \cdot x$ from that information. We refer to this mechanism as "creating polys from `type-set`."

Third, `type-set`'s reasoning abilities can be extended through the use of certain rules. ACL2 comes with some of these already built in. They include rules about the basic arithmetic functions such as that $x \cdot y$ is a positive rational if both x and y are. This rule can, via the above-mentioned mechanism of creating polys from `type-set`, provide a small amount of nonlinear reasoning to the linear arithmetic procedures. We shall see this shortly.

I.3.2 Linearization

Linearization is the process of converting an ACL2 term into one or more polys. We note the following:

[4]The order used here is based on a lexicographic one which considers the number of variables first, the number of function symbols second, and an alphabetical order last.

[5]Note that there are cancellations which can be performed.

[6]A category with only one member.

1. An equality can be expressed as a conjunction of two inequalities; $x = y$ is true if and only if both $x \leq y$ and $y \leq x$ are true.[7].

2. We normalize polys so that their leading coefficient is $+/-1$.

3. Consider the ACL2 term[8] `(< x y)`. If we know that both `x` and `y` are integers, we can assume that we are linearizing `(<= (+ x 1) y)` instead, and so convert `(< x y)` to the poly $0 \leq y + -1 \cdot x + -1$ rather than the weaker $0 < y + -1 \cdot x$. We shall refer to this as "the 1+ trick."

I.4 Linear Arithmetic

In this section, we describe the innermost loop of ACL2's arithmetic procedures. The linear arithmetic algorithm described in this section is a decision procedure for linear arithmetic over the rationals. We shall later refer to this algorithm as the linear arithmetic algorithm.

As in the examples of Section I.2, our goal is to derive a contradiction. In order to do so, all of the unknowns of a poly must be eliminated by cancellation. We can choose to eliminate the unknowns in any order, but we eliminate them largest unknowns first. That is, two polys are canceled against each other only when their largest unknowns are the same and have coefficients of opposite signs. Note that this occurs precisely when two polys are (or will be) in opposite sides of the same pot.

I.4.1 The Linear Arithmetic Algorithm

We start with a (possibly empty) pot-list and a list of polys to be added to it. We repeat the following until we reach a fixed point.

1. For each poly to be added, find its pot (the one whose label matches the poly's largest unknown), if there is one, or make a new one. Add the poly to this pot and cancel the new poly with any polys of the opposite sign. If this generates a contradiction, quit and return the contradiction; otherwise set any new polys aside for the moment.

2. If there were any new polys set aside in step 1, go back to step 1 with the new polys. Otherwise, go on to step 3.

3. For each pot that is new or has been changed by having polys added to it in step 1, see if we can create a poly from `type-set` about the label of the changed pot. Collect any such newly created polys, and return to step 1 with them.

Note that we are doing a breadth first search here.

[7]Note that the negation of an equality can similarly be expressed as a *disjunction* of two inequalities. We do not address this further in the present paper except to say that ACL2 does handle such situations.

[8]ACL2 terms are Lisp expressions, and therefore use a Lisp-style prefix syntax.

I.4.2 An Example

Suppose that we want to prove

$$\text{integer}\, a \quad \wedge \quad \text{integer}\, b \quad \wedge \quad 0 \leq a \quad \wedge \quad a < b \quad \Longrightarrow \quad a + 1 \leq a \cdot b + b$$

As before, we assume the hypotheses and the negation of the conclusion, and look for a contradiction. Since a and b are assumed to be integers, the linearization of $a < b$ is $0 \leq b + -1 \cdot a + -1$. Similarly $a \cdot b$ is known to be an integer since a and b are, so the linearization of the negation of $a + 1 \leq a \cdot b + b$ is $0 < -1 \cdot a \cdot b + -1 \cdot b + a$. Both of these linearizations used the 1+ trick. Finally, the linearization of $0 \leq a$ is just $0 \leq a$.

Since no cancellations can be performed between these three polys, executing steps 1 and 2 above results in the pot-list

label	positives	negatives
a	$0 \leq a$	
b	$0 \leq b + -1 \cdot a + -1$	
$a \cdot b$		$0 < -1 \cdot a \cdot b + -1 \cdot b + a.$

In step 3 we create three polys from `type-set`. There are three new (and therefore changed) pots, a, b, and $a \cdot b$. For the first two of these `type-set` knows that a and b are nonnegative integers, and so we create the polys $0 \leq a$ and $0 \leq b$. As mentioned in I.3.1 `type-set` therefore also knows that $a \cdot b$ is a nonnegative integer, and so we create the poly $0 \leq a \cdot b$. This small amount of nonlinear reasoning has long been built into ACL2. Note that we used the pot labels to guide our search for additional polys.

When we add these to the pot-list, after executing step 1 *once*, we get

label	positives	negatives
a	$0 \leq a$	
b	$0 \leq b$	
	$0 \leq b + -1 \cdot a + -1$	
$a \cdot b$	$0 \leq a \cdot b$	$0 < -1 \cdot a \cdot b + -1 \cdot b + a$

with the poly $0 < -1 \cdot b + a$ having been set aside. This poly is the result of cancelling the two polys in the $a \cdot b$ pot[9]. Upon adding it and canceling the polys in the b pot (executing steps 2 and 1 again), we get the contradiction $0 < -1$ and our lemma is proved.

I.5 Linear Lemmas

In versions of ACL2 previous to 2.7, the arithmetic procedure encompassed little more than what we describe in this section. It is still the standard behavior of

[9]Note that cancellation does not remove any polys. We augment, but never diminish, the pot list.

ACL2 when nonlinear arithmetic is disabled. Note that this procedure is not complete.

Suppose that the procedure described above does not produce a contradiction but instead yields a set of nontrivial polys. A contradiction might still be generated if we could add to the set some additional polys which allow further cancellation. That is where linear lemmas come in. When the set of polys has stabilized under the procedure described above and no contradiction has been produced, we form a list of the labels of any newly created pots and search the database of linear rules for ones that pattern match with a pot label. For each rule found, if we are able to relieve its hypotheses, we add its conclusion to the pot list (using the above linear arithmetic algorithm) in the hope that this will allow further cancellations to proceed. Just as for polys from `type-set`, we are using pot labels to guide our search for additional polys. Such labels, recall, correspond to the unknowns that are candidates for cancellation.

I.5.1 The Linear Lemmas Algorithm

As before, we start with a list of polys and a (possibly empty) pot list. We repeat the following until we reach a fixed point.

1. Add the polys with the linear arithmetic algorithm as described in section I.4.1; if no contradiction was generated, go on to step 2.

2. Make a list of the labels from any new pots created in step 1 (or 3). If there aren't any, quit and return the pot-list; otherwise, go on to step 3.

3. For each item in this list and for each applicable linear-lemma:
   If we can relieve the lemma's hypotheses, add the concluding poly(s) to the pot list as described in section I.4.1.

4. If a contradiction was generated, quit and return it. Otherwise, go back to step 2.

I.5.2 An Example

Suppose that we are given the following linear lemma, `expt-lemma`, about x^n

$$1 < x \quad \wedge \quad \text{integer}\, n \quad \wedge \quad 1 < n \quad \Longrightarrow \quad x < x^n,$$

and that we wish to prove

$$2 < x \quad \wedge \quad \text{integer}\, n \quad \wedge \quad 1 < n \quad \wedge \quad a \leq x + b \quad \Longrightarrow \quad a < x^n + b.$$

After linearizing the inequalities among the hypotheses and the negation of the conclusion and adding them to the empty pot-list (step 1) we get

label	positives	negatives
b		$0 < -1 \cdot b + a$
n	$1 < n$	
x	$0 \leq x + b + -1 \cdot a$	
	$0 < x + -2$	
x^n	$0 < x^n$	$0 \leq -1 \cdot x^n + -1 \cdot b + a$

Note that the poly $0 < x^n$ was created from `type-set` about the pot-label x^n.

In step 2, we note that there were four new pots created in step 1, and in step 3 we will eventually find `expt-lemma`. We sketch here how we relieve the first hypothesis, $1 < x$. Rewriting cannot do anything with this, so we linearize the negation of the hypothesis (yielding $0 \leq -1 \cdot x + 1$) and recursively call the very procedure we are describing. Here is a situation where we do not start with an empty pot list. This poly will be added to the x pot. Upon cancellation with $0 < x + -2$, we get the contradiction $0 < -1$, and the hypothesis has been relieved.

We therefore add the linearization of the conclusion of `expt-lemma`, $0 < x^n + -1 \cdot x$, to the pot-list. After a couple of rounds of cancellation we derive the contradiction $0 < -2$, and the theorem has been proved.

I.6 Linear Lemmas Revised

When nonlinear arithmetic is enabled, we do the above procedure a little differently. The gathering of polys from linear lemmas is intended to let the process of cancellation continue. In the procedure described in this section we still use linear lemmas, but we intertwine their use with other ways of gathering polys in preparation for what is to come — the nonlinear arithmetic procedure.

I.6.1 Exploded Pot Labels

Previously, we examined pot labels to direct our gathering of additional polys from such sources as `type-set` and linear lemmas. That is, when there was a pot labeled with, say x^n, we looked to `type-set` or linear lemmas for additional information about x^n. We shall soon examine "exploded" pot labels. These exploded pot labels consist of the original pot label and, if the pot label is a product, each of the label's factors. A few examples will make this clearer:

- $x \implies x$
- $\lfloor x \rfloor \implies \lfloor x \rfloor$
- $x \cdot \lfloor x \rfloor \implies x$, $\lfloor x \rfloor$, and $x \cdot \lfloor x \rfloor$
- $x \cdot y \cdot z \implies x$, y, z, and $x \cdot y \cdot z$

We are doing this so that we can seed the database with information about the factors of products. Note that in the last example we do not examine, for instance, $x \cdot y$.

I.6.2 Inverse Polys and Bounds Polys

Division introduces additional issues. We represent the ratio x/y as $x \cdot \frac{1}{y}$. A term is said to "involve division" if it is of the form

1. $\frac{1}{x}$ or $(\frac{1}{x})^n$, or
2. x^c or $x^{c \cdot n}$, where c is a constant negative integer.

We also say that a term or pot label involves division if any of its factors do.

As preparation for the nonlinear procedure, given such a term about division, ACL2 "adds its inverse polys." We do not attempt to describe the method for generating these polys here, but give a few examples:

- If we can determine that $4 < x$, we know both $0 < \frac{1}{x}$ and $\frac{1}{x} < 1/4$.
- If we can determine that $0 < \frac{1}{x}$ and $\frac{1}{x} < 3$, we know $1/3 < x$.
- If we can only determine that $-2 < x$, we do not know anything about $\frac{1}{x}$.

This information will be used by deal-with-division, described below.

A "bounds poly" is a poly with exactly one unknown. These polys can be considered to bound the unknown. For instance, the poly $0 < x + 1$ can be considered to give a lower bound of -1 for x. Similarly, $0 < -1 \cdot x + 3$ bounds x from above at 3. A term is said to have "good bounds" if there are bounds polys for that term which bound the term away from zero. This will become important later when we are multiplying certain polys together. For example, we may wish to multiply the two polys, $0 < x \cdot \frac{1}{y}$ and $0 < y$, to form the new poly $0 < x$. But, since this requires rewriting $y \cdot \frac{1}{y}$ to 1, it can be done only if y is known to be non-zero.

I.6.3 The Revised Linear Lemmas Algorithm

As mentioned above, when nonlinear arithmetic is enabled we do things a bit differently. As before, we start with a list of polys and a (possibly empty) pot list. We repeat the following until we reach a fixed point.

1. Add the polys with the linear arithmetic algorithm as described in section I.4.1; if no contradiction was generated, go on to step 2.
2. Make an exploded list of the labels of any new pots. If there are not any, quit and return the new pot-list. Otherwise, go on to step 3.
3. For each item in this list:
   (a) Add any polys created from `type-set`.
   (b) For each applicable linear lemma: If we can relieve its hypotheses, add the concluding poly(s) to the pot-list as in section I.4.1.
   (c) If the item involves division, add any inverse polys
4. If a contradiction was generated, quit and return it. Otherwise, go back to step 2.

I.6.4 An Example — Part I

Let us consider $0 < a \quad \wedge \quad a < b \quad \Longrightarrow 1 < b/a$. After adding the initial polys, the pot-list will look like

label	positives	negatives
a	$0 < a$	
b	$0 < b + -1 \cdot a$	
$b \cdot \frac{1}{a}$		$0 \leq -1 \cdot b \cdot \frac{1}{a} + 1$

In step 2, we make the list a, b, $\frac{1}{a}$, $b \cdot \frac{1}{a}$. Note the presence of $\frac{1}{a}$, which would not have been there if we used regular pot labels. In step 3a, we create the poly $0 < \frac{1}{a}$ from `type-set` and add it to the pot[10]. We will continue this example in Section I.7.1.

I.7 Nonlinear Arithmetic

Before proceeding, let us pause a moment to recollect where we are. In Section I.4.1, we presented the linear arithmetic algorithm which lies at the heart of ACL2's arithmetic procedures — both linear and nonlinear. We next described the previously existing linear lemmas algorithm in Section I.5.1. This algorithm uses the linear arithmetic algorithm and is still the default behavior when nonlinear arithmetic is not enabled. Next, in Section I.6.3 we described a variant of the linear lemmas algorithm which is used when nonlinear arithmetic is enabled. Whereas the previously existing linear lemmas algorithm is the outermost loop for arithmetic reasoning when nonlinear arithmetic is disabled, the new variant is only the middle loop when nonlinear arithmetic is enabled. We are now about to describe the outermost loop of the nonlinear arithmetic algorithm.

The nonlinear arithmetic procedure consists of three subprocedures — deal-with-product, deal-with-factor, and deal-with-division. Each of these subprocedures is guided by pot-labels and attempts to multiply polys together. In order to multiply two polys, we unlinearize the polys (converting them back into ACL2 terms), create the term representing their product, use general-purpose rewriting to rewrite the product terms, and linearize the result. For example, the product of the two polys $0 < -1 \cdot x + 3$ and $0 \leq y + a$ is $0 \leq -1 \cdot y \cdot x + -1 \cdot x \cdot a + 3 \cdot y + 3 \cdot a$. In order to multiply two pots together, form a list of the polys in each pot and multiply each poly in the first list with each poly in the second. We multiply more than two polys or pots together by generalizing the above.

I.7.1 Deal-with-product and Deal-with-factor

When we have polys about a product and we have polys about the product's factors, we can multiply those polys about the factors to form polys about the product and perhaps thereby allow cancellation to proceed.

[10] We also create the same poly in step 3c

For instance, if we have a new pot about the product $a \cdot b \cdot c$, we can form new polys by finding pots with any of the following combinations of labels and then multiplying the appropriate pots together.

- a, b, and c
- a, and $b \cdot c$
- $a \cdot c$, and b
- $a \cdot b$, and c

This is done by the subprocedure deal-with-product.

Similarly, if the new pot is about a, we look for pots of which a is a factor, such as $a \cdot b \cdot c$, and then see if we can complete the product. This is done by the subprocedure deal-with-factor. We use these two procedures in tandem so that we are less sensitive to the order in which pots are created.

Let us revisit the example from I.6.3. When we left it, having just added the poly from `type-set`, it looked like

label	positives	negatives
a	$0 < a$	
$\frac{1}{a}$	$0 < \frac{1}{a}$	
b	$0 < b + -1 \cdot a$	
$b \cdot \frac{1}{a}$		$0 \leq -1 \cdot b \cdot \frac{1}{a} + 1$

We can now use either deal-with-product or deal-with-factor on the three pots $\frac{1}{a}$, b, and $b \cdot \frac{1}{a}$. Multiplying $0 < \frac{1}{a}$ and $0 < b + -1 \cdot a$, we get $0 < b \cdot \frac{1}{a} + -1 \cdot a \cdot \frac{1}{a}$. But, since a is known to be non-zero, this will be rewritten to

$$0 < b \cdot \frac{1}{a} + -1.$$

Upon adding this to the pot-list we get the contradiction $0 < 0$, and we are done.

I.7.2 Deal-with-division

Let us next consider $0 < b \quad \wedge \quad b < a \quad \Longrightarrow 1 < a/b$ After executing the revised linear lemmas algorithm, the pot-list will look like

label	positives	negatives
a	$0 < a$	
b	$0 < b$	$0 < -1 \cdot b + a$
$\frac{1}{b}$	$0 < \frac{1}{b}$	
$a \cdot \frac{1}{b}$		$0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$

This time, deal-with-product and deal-with-factor are insufficient. If we multiply $0 < a$ and $0 < \frac{1}{b}$, we get $0 < a \cdot \frac{1}{b}$; but when canceled with $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$ we only get the trivially true poly $0 < 1$ which does us no good. Rather, we

want to multiply the polys $0 < b$ and $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$. After rewriting $a \cdot b \cdot \frac{1}{b}$ to a, we have $0 < b + -1 \cdot a$. Upon adding this latter poly to the pot-list, we get $0 < 0$ and the lemma is proved.

We now sketch the algorithm behind deal-with-division.

1. If the current pot label being examined is itself a product, quit. If we have good bounds for the label, go to step 2; if not, quit.

2. Make a list of all the pot labels that have the multiplicative inverse of the current label as a factor. To distinguish them from the current pot, we will refer to the pots these labels belong to as the "found" pots. For each entry in this list:

   (a) Multiply the bounds polys from the current pot with the polys in the found pot.

   (b) Multiply the bounds polys from the found pot with the bounds polys from the current pot.

Let us see how this lines up with what we said we wanted to do above. When deal-with-division is examining the pot-label b (which has good-bounds) it finds the pot-labels $\frac{1}{b}$ and $a \cdot \frac{1}{b}$. Since $0 < b$ and $0 \leq -1 \cdot a \cdot \frac{1}{b} + 1$ are both bounds polys, we multiply these polys together in step 2. As above, upon adding this latter poly to the pot-list, we get $0 < 0$ and the lemma is proved.

I.7.3 The nonlinear Arithmetic Algorithm

After adding polys as in Section I.4.1, loop through the following at most three times. If at any point we generate a contradiction, quit and return it.

1. Execute the revised linear lemmas algorithm, described in Section I.6.3.

2. Make a list of the labels from any new pots and for each item in that list:

   (a) If we have good-bounds for the current item, carry out deal-with-division and add any polys generated.

   (b) If the current item is a product, carry out deal-with-product and add any polys generated.

   (c) If the current item is not a product, carry out deal-with-factor and add any polys generated.

This concludes our presentation of ACL2's nonlinear arithmetic algorithm.

I.8 Conclusion

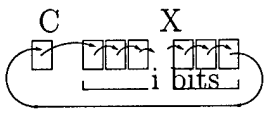
The nonlinear arithmetic procedure is tightly integrated with the rest of ACL2 and allows lemmas such as the ACL2 equivalent of the following to be proven automatically.

$$(\mathrm{bc}\, i\, j) = \frac{i!}{j!(i-j)!}$$

Where bc is the binomial coefficient defined by:

```
(defun bc (i j)
  (if (< i 0)
      0
    (if (< i j)
        0
      (if (<= j 0)
          1
        (+ (bc (1- i) j)
           (bc (1- i) (1- j)))))))
```

We can also prove the next lemma automatically. It states that rotating right an i-bit wide register through a carry flag fits back into the i-bit wide register.




```
(defthm example-lemma-2
    (implies (and (integerp i)
                  (integerp x)
                  (< x (expt 2 i))
                  (or (equal c 0)
                      (equal c 1)))
             (< (+ (floor x 2)
                   (* (expt 2 (+ -1 i)) c))
                (expt 2 i))))
```

Our nonlinear arithmetic extension to ACL2 provides significant benefits at a small cost. Proofs that do not involve any nonlinear inequalities are not effected and run at the same speed. A typical "small" lemma with a couple of nonlinear inequalities which ACL2 could prove automatically before, will generally be proven within a few percentage points of the time previously required — but we can now automatically prove more of these. For more complicated lemmas and theorems, little can be said about the computer time required.

However, within broad limits, the time a user takes to complete a proof is of greater importance than the time the computer takes. Examining failed proofs and writing helper lemmas can be time consuming and psychologically

draining. The fewer lemmas a user has to prove on the way to a desired result, the better. One example of the power of the nonlinear procedure is the proof that a highly optimized 6502 assembly language program multiplies two numbers correctly. Previous versions of this proof required two dozen or more helper lemmas to be proven; we can now do the proof with only five lemmas. The lemma `example-lemma-2` above is one of these.

Bibliography

[1] A. Armando and S. Ranise. A Practical Extension Mechanism for Decision Procedures. Journal of Universal Computer Science, Volume 7, Issue 2, pp. 124-140, February 2001.

[2] R. Boyer and J Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. Machine Intelligence, Volume 11, pp. 83-124, 1988.

[3] D. Cyrluk and D. Kapur. Reasoning about Nonlinear Inequality Constraints: A Multilevel Approach. Proceedings DARPA workshop on Image Understanding, 1989, pp. 904-915.

[4] J. Harrison. Theorem Proving with the Real Numbers. Technical Report TR-408, University of Cambridge Computer Laboratory, December 1996.

[5] J. Harrison. Verifying the Accuracy of Polynomial Approximations in HOL. Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'97, Springer LNCS 1275, pp. 137-152.

[6] D. Kapur. A Rewrite Rule Based Framework for Combining Decision Procedures. Frontiers of Combining Systems, A. Armando, editor. Volume 2309 of Lecture Notes in Computer Science, pp. 87-102, Springer, 2002.

[7] M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, 2000.

[8] M. Kaufmann, P. Manolios, and J Moore. Editors. Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, 2000.

[9] M. Kaufmann and J Moore. ACL2: An Industrial Strength Version of Nqthm. Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96), pp. 23-34, IEEE Computer Society Press, June 1996.

[10] P. Miner and J. Leathrum. Verification of IEEE Compliant Subtractive Division Algorithms. Formal Methods in Computer-Aided Design (FMCAD), Volume 1166 of Lecture Notes in Computer Science, pp. 64-78.

[11] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 1, Issue 2, October 1979, pp. 245-257. Springer, 1996.

[12] H. Rueß and N. Shankar. Combining Shostak Theories. Proccedings of RTA 2002, LNCS 2378, pp. 1-18. Springer-Verlag, 2002.

[13] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions. LMS Journal of Computation and Mathematics, Volume 1, pp. 148-200, December, 1998.

[14] R. Shostak. Deciding Combinations of Theories. Journal of the ACM (JACM), Volume 31, Issue 1, January 1984, pp. 1-12.